

EECS402 Lecture 13

Andrew M. Morgan

Savitch Ch. 8
Operator Overloading
Returning By Constant Value

1

Consider This Program 402 class ChangePocketClass int getDimes() public: return dimes; ChangePocketClass():quarters(0),dimes(0) private: ChangePocketClass(int q, int d): int quarters; quarters(q), dimes(d) int dimes; **(**) }; void setQuarters(int val) From main: quarters = val; ChangePocketClass c1; ChangePocketClass c2; void setDimes(int val) ChangePocketClass c3; c1.setQuarters(5); dimes = val; c1.setDimes(7); c2.setQuarters(3); int getQuarters() c2.setDimes(8); return quarters; c3 = c1 + c2;EECS Andrew M Morgan 402



Program Discussion

403

- What does the line "c3 = c1 + c2;" do?
 - Logically, you would expect it to add the number of quarters in the c1
 object to the number of quarters in the c2 object, and add the dimes as
 well
- Would you expect that adding two objects of any class works this way? (A member-by-member addition?)
 - No
 - For example, adding two "PersonClass" objects together doesn't make sense
- Due to this, addition is not provided by default for user-defined classes
 - Therefore, the compiler will not allow the addition (yet)

EECS 402

Andrew M Morgan

M

3



Using Operators

403 EECS

- While the operator "+" makes perfect sense for types such as int and float, what does it mean for user-defined data types?
 - The programmer must define it, and implement it when appropriate
 - Often, it doesn't make any sense to perform mathematical operations on user-defined class types
- Rather than use the "+" operator, programmer could write a member function named "add":
 - ChangePocketClass ChangePocketClass::add(const ChangePocketClass &rhs) const;
- Users aren't used to writing "c3 = c1.add(c2);" to add two items
 - "c3 = c1 + c2;" would make a lot more sense, and wouldn't require a user to look up a function prototype to determine how to add two objects together

EECS 402

Andrew M Morgan

M

Using Operators With User-Defined Data Types



- C++ allows a programmer to customize common operators work for different classes
- When C++ comes across an operator for user-defined data types, a function is called to perform the operation
 - The "+" operator results in a call to a function named "operator+"
 - The "=" operator results in a call to a function named "operator="
 - etc
- The following list of operators can be overloaded:

```
+ - * / % ^ & | ~
! = < > += -= *= ?= %=
^= &= |= << >> >>= <= == !=
<= >= && || ++ -- ->* , ->
[] () new delete new[] delete[]
```

 Note: Just because you can overload an operator doesn't necessarily mean it is a good idea

EECS 402

Andrew M Morgan



5

Overloading Operators, Example class ChangePocketClass public: ChangePocketClass(): quarters(0), dimes(0) ChangePocketClass(int q, int d): quarters(q), dimes(d) ...no change to readers/writers ChangePocketClass operator+(const ChangePocketClass in) ChangePocketClass result; result.quarters = quarters + in.quarters; result.dimes = dimes + in.dimes; return result; private: int quarters; int dimes; EECS Andrew M Morgan 402

```
Output Of Main
 int main()
   ChangePocketClass c1;
   ChangePocketClass c2;
   ChangePocketClass c3;
   c1.setQuarters(5);
   c1.setDimes(7);
   c2.setQuarters(3);
   c2.setDimes(8);
   c3 = c1 + c2;
   cout << "c1 q: " << c1.getQuarters() << " d: " << c1.getDimes() << endl;</pre>
   cout << "c2 q: " << c2.getQuarters() << " d: " << c2.getDimes() << endl;
   cout << "c3 q: " << c3.getQuarters() << " d: " << c3.getDimes() << endl;</pre>
   return 0:
                                    c1 q: 5 d: 7
                                    c2 q: 3 d: 8
                                    c3 q: 8 d: 15
EECS
                                  Andrew M Morgan
402
```



Overloading The Assignment Operator

402 402

- You always get an operator=() function by default.
 - If you do not overload it, it will do a member-by-member copy, just like the default copy ctor
 - Any problems that occur due to the copy ctor will also be problems for operator= (will discuss further during dynamic allocation discussion)
- The following is a general syntax for operator=():

```
ClassName ClassName::operator=(const ClassName Crhs)
```

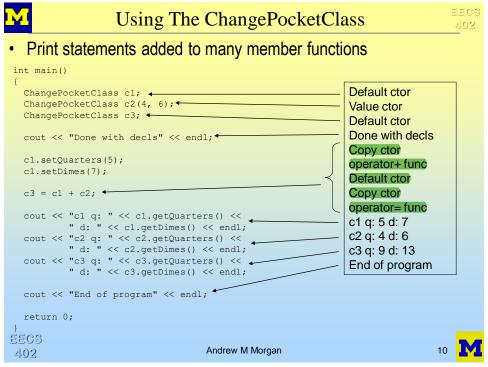
- Note: Returns "ClassName&" to allow "nested assignment", such as:
 val1 = val2 = val3; //val1 assigned to the evaluation of the "val2 = val3" expression
- Can also just return "void" but then nested assignment won't work
- rhs is an abbreviation for "right hand side" since the parameter is the object on the rhs of the equal sign

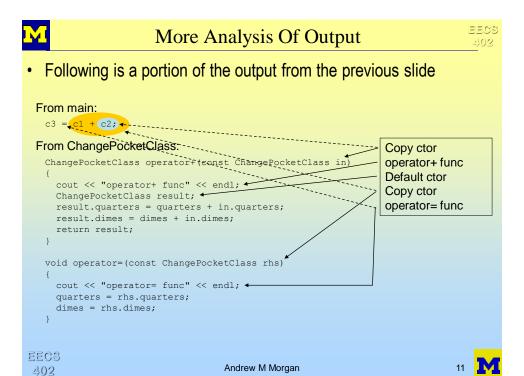
EECS 402

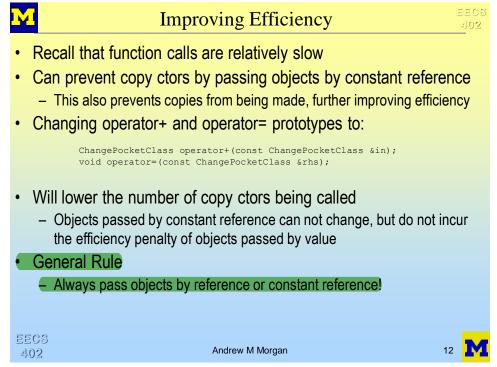
Andrew M Morgan



```
Adding Assignment Operator and Copy Ctor
class ChangePocketClass
  public:
    //rest of member functions as shown before
    //Copy constructor-called when a copy of an object is needed
    ChangePocketClass(const ChangePocketClass &copy)
      quarters = copy.quarters;
      dimes = copy.dimes;
                                      I do not need to write this right?
    //Assignment operator-called when one object is assigned to another
    void operator=(const ChangePocketClass rhs)
                                          why no & here?
      quarters = rhs.quarters;
      dimes = rhs.dimes;
  private:
    int quarters;
    int dimes;
};
EECS
                               Andrew M Morgan
402
```









Optional Topic: Increment / Decrement Oddity

4)02

- There are actually two versions of the increment and decrement operators
 - Postfix version: The "++" comes after the variable
 - In this case, the variable is still incremented, but the expression evaluates to the value before the increment occurred
 - Prefix version: The "++" comes before the variable
 - In this case, the variable is incremented, and the expression evaluates to the incremented value

Example:

```
int int1, int2;
int1 = 10;
int2 = int1++;
cout << "POST: int1 was 10 - after int1: " << int1 << " int2: " << int2 << end1;
int1 = 10;
int2 = ++int1;
cout << "PRE: int1 was 10 - after int1: " << int1 << " int2: " << int2 << end1;</pre>
```

Output:

```
POST: int1 was 10 - after int1: 11 int2: 10

PRE: int1 was 10 - after int1: 11 int2: 11

The difference is in what the increment expression evaluates to

int1 is always incremented as expected, regardless of postfix or prefix
```

Andrew M Morgan

13

EECS

402



Overloading Unary Operators

403 403

- Not all operators have two operands (like + and *)
 - Operators that operate on a single operand are called "unary operators"
 - Examples: -myVar (negate operator), val++ (postfix increment operator),
 ++val (prefix increment operator), etc
- Unary operators normally don't require any parameters, since they only operate on one object (i.e. one operand)
- For increment and decrement operators, postfix and prefix versions typically act differently
 - Both increment operators are named "operator++"
 - To differentiate between versions, by convention, the postfix operator takes in a dummy int parameter that has no meaning
 - The int parameter is used only to cue the language that the operator is postfix, rather than prefix

EECS 402

Andrew M Morgan

M

```
An IntClass With Many Operators
                                               //Prefix version (i.e. ++myObj)
class IntClass
                                               //increments the value first, and
                                               //the expression evaluates to the
  int val; //The value of the integer
                                               //incremented value
public:
                                               const IntClass operator++()
  //Print out attrs of the object
  void print() const
                                                 IntClass retVal;
                                                 val++;
    cout << "Val: " << val << endl;
                                                 retVal.val = val;
                                                 return retVal;
  //Assign to an integer var
                                               //Postfix version (i.e. myObj++)
  void operator=(int iVal)
                                               //increments the value, but the
    val = iVal;
                                               //expression evaluates to the value
                                               //before the increment occurred
                                               const IntClass operator++(int dummy)
  //Negate operator. Note: this operator
  //has no "side-effects". In other
                                                 IntClass retVal;
  //words, it doesn't change the object
                                                 retVal.val = val;
  //is operating on.
                                                 val++;
  const IntClass operator-() const
                                                 return retVal;
    IntClass retVal;
                                            };
    retVal.val = -val;
    return retVal;
EECS
                                   Andrew M Morgan
402
```

```
Using The IntClass
                                                                                         4)02
  int main()
                                                              iObj1 during increment:
                                                              Val: 14
    IntClass iObj1;
                                                              iObj2 during increment:
    IntClass iObj2;
    iObj1 = 14;
                                                              iObj1 after increment:
    iObj2 = iObj1;
                                                              Val: 15
    cout << "iObj1 during increment:" << endl;</pre>
                                                              iObj2 after increment:
    (iObj1++).print();
cout << "iObj2 during increment:" << endl;</pre>
                                                              Val: 15
                                                              iObj1 after negate:
    (++iObj2).print();
    cout << "iObj1 after increment:" << endl;</pre>
                                                              Val: -15
    iObj1.print();
                                                              iObj2 after negate:
    cout << "iObj2 after increment:" << endl;</pre>
                                                              Val: 15
    iObj2.print();
    iObj1 = -iObj2;
    cout << "iObj1 after negate:" << endl;</pre>
    iObj1.print();
    cout << "iObj2 after negate:" << endl;</pre>
    iObj2.print();
    return 0;
  }
EECS
                                       Andrew M Morgan
402
```



Operators As Non-Members

403 EEC

- All overloaded operators so far have been member functions
- Binary operators (such as x * y) have two operands, but only one parameter is passed in
 - The left hand side operand is the object that the function was called on, and the right hand side operand is passed in
 - For example, consider "iObj1 * 4"
 - What if the left hand side operand isn't an object?
 - For example, consider "4 * iObj1"
 - Users are used to both versions being available, but they are actually two completely different functions
 - Since the left hand side of the second version is not an object, this funtion can not be a member function
 - Must be made a global function with two parameters

EECS 402

Andrew M Morgan

17



17

```
New Operators For IntClass
                        ass IntClass
                          //Previously described members here...
                                                                         From main():
                           //Reader funtion for val
                                                                           iObj1 = 5;
                          int getVal() const
                                                                           cout << "iObj1 * 4:" << endl;
                            return val;
                                                                           (iObj1 * 4).print();
                                                                           cout << "4 * iObj1:" << endl;
                                                                           (4 * iObj1).print();
                           //This function allows for
                           //myObj * 4 multiplication
                          const IntClass operator*(int rhs) const
                            IntClass retVal;
                            retVal.val = val * rhs;
                            return retVal;
                                                                                     iObi1 * 4:
                                                                                     Val: 20
This is a global function!
                                                                                     4 * iObj1:
                       //This fucntion allows for "4 * myObj" multiplication
                                                                                     Val: 20
                       const IntClass operator*(int lhs, const IntClass &rhs)
                         IntClass retVal;
                         retVal = lhs * rhs.getVal();
                       ÉECS
                                                          Andrew M Morgan
                       402
```



Overloading The Insertion Operator (<<)

403 403

- The << operator is called the insertion operator
 - It is used to "insert" values into an output stream
- Consider "cout << val << endl:"
 - "cout << val" is performed first
 - Then the insertion operator is used again, with "endl" on the right side
 - What is on the left side of the operator, though?
 - Answer: The left side is the return value of "cout << val"
- When the insertion operator is used, it modifies the stream object
 - Attributes regarding the number of characters in the stream, etc, must be changed when the insertion operator is used
 - Since the insertion operator can be used multiple times as shown above, the insertion operator must return a reference, so the returned value can be changed during subsequent calls to the insertion operator

EECS 402

Andrew M Morgan

19



19



Specifics For operator<<

402 402

- The cout object in iostream is of type "ostream"
- Consider "cout << iObj1;"
 - The left side is of type ostream, and the right side is of a class type
 - It must return a reference, as described earlier
- Prototype for overloading the insertion operator for the IntClass would, therefore, be as follows:

What does the & mean here?

ostream operator << (ostream os, const IntClass &iObj);

return by reference!

- Note: This function can **not** be a member function
 - As described earlier, the left hand side operand is not an IntClass object
 - It could be a member function of the ostream class, but you don't have the ability to add members to the ostream class
 - Therefore, it must be a global function in the program

EECS 402

Andrew M Morgan

0



```
Implementation Of operator<</p>

ostream& operator<<(ostream &os, const IntClass &iObj)
{
os < iObj.getVal();
return os;
}

From main():

iObj1 = 81;
cout << "iObj1: " << iObj1 << endl;
iObj2 = iObj1++;
cout << "iObj1: " << iObj1 << " iObj2: " << iObj2 << endl;

iObj1: 81
iObj1: 81
iObj1: 82 iObj2: 81

EEGS
402

Andrew M Morgan
21

Andrew M Morgan
21

Andrew M Morgan
21

Ostream& operator<</pre>
```





More On Insertion Operator

403

- Thanks to the property of inheritance, an ofstream can be passed into a function expecting an ostream
 - Details of inheritance won't be discussed here
- The insertion operator written on the last slide can be used to output to files in addition to the cout object

```
ofstream outFile;
                                          when to .c_str()?
        outFile.open("testOut.txt");
        iObj1 = 81;
        outFile << "iObj1: " << iObj1 << endl;
        iObj2 = iObj1++;
        outFile << "iObj1: " << iObj1 << " iObj2: " << iObj2 << endl;
        outFile.close();
  Contents of testOut.txt after execution:
                                            Note: No change was necessary to
     iObj1:81
                                            operator<< function to allow this
     iObj1: 82 iObj2: 81
                                            functionality
EECS
                                 Andrew M Morgan
402
```