The University Of Michigan

# EECS402 Lecture 13

Andrew M. Morgan

Savitch Ch. 8
Operator Overloading
Returning By Constant Value

1

## Consider This Program



```
class ChangePocketClass
{
  public:
    ChangePocketClass():quarters(0),dimes(0)
    {}
    ChangePocketClass(int q, int d):
                  quarters(q), dimes(d)
    {}
    void setQuarters(int val)
    {
      quarters = val;
    }
    void setDimes(int val)
    {
      dimes = val;
    }
    int getQuarters()
    {
      return quarters;
    }
    int getDimes()
    {
      return dimes;
    }
  private:
    int quarters;
    int dimes;
};
```

From main:

```
ChangePocketClass c1;
ChangePocketClass c2;
ChangePocketClass c3;
c1.setQuarters(5);
c1.setDimes(7);
c2.setQuarters(3);
c2.setDimes(8);

c3 = c1 + c2;
```



2

## Program Discussion



- 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)





## Using Operators



- 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

## 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





## 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;
};
```

## 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;
  cout << "c2 q: " << c2.getQuarters() << " d: " << c2.getDimes() << endl;
  cout << "c3 q: " << c3.getQuarters() << " d: " << c3.getDimes() << endl;

  return 0;
}
```

```
c1 q: 5 d: 7
c2 q: 3 d: 8
c3 q: 8 d: 15
```



## Overloading The Assignment Operator



- 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 &rhs)
```

  - 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

# 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;
    }

    //Assignment operator-called when one object is assigned to another
    void operator=(const ChangePocketClass rhs)
    {
      quarters = rhs.quarters;
      dimes = rhs.dimes;
    }

  private:
    int quarters;
    int dimes;
};
```





# Using The ChangePocketClass



- Print statements added to many member functions

```
int main()
{
  ChangePocketClass c1;
  ChangePocketClass c2(4, 6);
  ChangePocketClass c3;

  cout << "Done with decls" << endl;

  c1.setQuarters(5);
  c1.setDimes(7);

  c3 = c1 + c2;

  cout << "c1 q: " << c1.getQuarters() <<
          " d: " << c1.getDimes() << endl;
  cout << "c2 q: " << c2.getQuarters() <<
          " d: " << c2.getDimes() << endl;
  cout << "c3 q: " << c3.getQuarters() <<
          " d: " << c3.getDimes() << endl;

  cout << "End of program" << endl;

  return 0;
}
```

Default ctor
Value ctor
Default ctor
Done with decls
Copy ctor
operator+ func
Default ctor
Copy ctor
operator= func
c1 q: 5 d: 7
c2 q: 4 d: 6
c3 q: 9 d: 13
End of program

# More Analysis Of Output



- Following is a portion of the output from the previous slide

From main:

```
c3 = c1 + c2;
```

From ChangePocketClass:

```
ChangePocketClass operator+(const ChangePocketClass in)
{
  cout << "operator+ func" << endl;
  ChangePocketClass result;
  result.quarters = quarters + in.quarters;
  result.dimes = dimes + in.dimes;
  return result;
}

void operator=(const ChangePocketClass rhs)
{
  cout << "operator= func" << endl;
  quarters = rhs.quarters;
  dimes = rhs.dimes;
}
```

Copy ctor
operator+ func
Default ctor
Copy ctor
operator= func





# Improving Efficiency



- Recall that function calls are relatively slow
- Can prevent copy ctors by passing objects by constant reference
  - This also prevents copies from being made, further improving efficiency
- Changing operator+ and operator= prototypes to:

```
ChangePocketClass operator+(const ChangePocketClass &in);
void operator=(const ChangePocketClass &rhs);
```

- Will lower the number of copy ctors being called
  - Objects passed by constant reference can not change, but do not incur the efficiency penalty of objects passed by value
- General Rule
  - Always pass objects by reference or constant reference!

# Optional Topic: Increment / Decrement Oddity



- 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 << endl;

int1 = 10;
int2 = ++int1;
cout << "PRE: int1 was 10 - after int1: " << int1 << " int2: " << int2 << endl;
```

- 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





# Overloading Unary Operators



- 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

# An IntClass With Many Operators



```
class IntClass
{
 private:
   int val; //The value of the integer
 public:
   //Print out attrs of the object
   void print() const
   {
     cout << "Val: " << val << endl;
   }

   //Assign to an integer var
   void operator=(int iVal)
   {
     val = iVal;
   }

   //Negate operator.  Note: this operator
   //has no "side-effects".  In other
   //words, it doesn't change the object
   //is operating on.
   const IntClass operator-() const
   {
     IntClass retVal;
     retVal.val = -val;
     return retVal;
   }

   //Prefix version (i.e. ++myObj)
   //increments the value first, and
   //the expression evaluates to the
   //incremented value
   const IntClass operator++()
   {
     IntClass retVal;
     val++;
     retVal.val = val;
     return retVal;
   }

   //Postfix version (i.e. myObj++)
   //increments the value, but the
   //expression evaluates to the value
   //before the increment occurred
   const IntClass operator++(int dummy)
   {
     IntClass retVal;
     retVal.val = val;
     val++;
     return retVal;
   }
};
```



15

# Using The IntClass



```
int main()
{
  IntClass iObj1;
  IntClass iObj2;

  iObj1 = 14;
  iObj2 = iObj1;

  cout << "iObj1 during increment:" << endl;
  (iObj1++).print();
  cout << "iObj2 during increment:" << endl;
  (++iObj2).print();
  cout << "iObj1 after increment:" << endl;
  iObj1.print();
  cout << "iObj2 after increment:" << endl;
  iObj2.print();

  iObj1 = -iObj2;
  cout << "iObj1 after negate:" << endl;
  iObj1.print();
  cout << "iObj2 after negate:" << endl;
  iObj2.print();

  return 0;

}
```

```
iObj1 during increment:
Val: 14
iObj2 during increment:
Val: 15
iObj1 after increment:
Val: 15
iObj2 after increment:
Val: 15
iObj1 after negate:
Val: -15
iObj2 after negate:
Val: 15
```



16

## Operators As Non-Members



- 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





## New Operators For IntClass



```
class IntClass
{
    //Previously described members here...

    //Reader funtion for val
    int getVal() const
    {
      return val;
    }

    //This function allows for
    //myObj * 4 multiplication
    const IntClass operator*(int rhs) const
    {
      IntClass retVal;
      retVal.val = val * rhs;
      return retVal;
    }
};

//This fucntion allows for "4 * myObj" multiplication
const IntClass operator*(int lhs, const IntClass &rhs)
{
  IntClass retVal;
  retVal = lhs * rhs.getVal();
  return retVal;
}
```

From main():

```
iObj1 = 5;

cout << "iObj1 * 4:" << endl;
(iObj1 * 4).print();
cout << "4 * iObj1:" << endl;
(4 * iObj1).print();
```

```
iObj1 * 4:
Val: 20
4 * iObj1:
Val: 20
```

## Overloading The Insertion Operator (<<)



- 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





## Specifics For operator<<



- 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:

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

- 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

## Implementation Of operator<<



```
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: 82 iObj2: 81
```





## More On Insertion Operator



- 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;
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:
iObj1: 81
iObj1: 82 iObj2: 81

Note: No change was necessary to operator<< function to allow this functionality