Lecture 07: Operator Overloading

We have already established that inheritance, polymorphism, and code reuse are fundamental themes to OOP. This lecture commences the exploration of polymorphism, while lecture 8 explores inheritance. Code reuse will be promoted and illustrated in each lecture (from this point onwards in the course). Bear in mind however, the two concepts often go hand in hand and are therefore mutual compliments.

C++ facilitates polymorphism in various ways, including the following approaches:

- Function overloading (already discussed in lecture 4))
- Function default arguments (also discussed in lecture 4)
- Operator overloading (discussed in this lecture)
- Function overriding (covered in lecture 8)
- Templates (covered in lecture 10)
- Object casting

Note: In some C++ circles where technical quibbling abounds, functional overloading, operator overloading, and templates (to be discussed in lecture 10) are not regarded as polymorphism, since they are resolved at compile-time, whereas function overriding and object casting are resolved at run-time. These pundits argue that polymorphism relates only to things resolved at run-time. The alternate view supported by this course is to view polymorphism in the true sense of the term, as being inclusive of *compile-time polymorphism* (including function overloading, operator overloading, and templates) as well as *run-time polymorphism* (including object casting and function overriding).

This lecture focuses on operator overloading. The lecture proceeds under the following captions:

- Introduction
- Overloading Binary Operators
- Overloading Unary Operators
- Friend Operator Functions
- Overloading Relational Operators
- Summary and Concluding Remarks

7.1 Introduction

One very powerful feature of C++ is the facility to redefine the meaning of an operator, when used with respect to an instance of a class. This is referred to as *operator overloading*. In operator overloading, the meaning of the operator is changed only when used with objects of the class in question; outside of this scenario, its original meaning holds.

To illustrate, the **+** operator may be overloaded to mean, push a new object to a stack, or add a new object to a queue, or insert a new node in a linked list, or add the corresponding components of two complex objects defined on the same class.

To overload an operator, you must define an *operator function* relative to the class to which the overloaded operator will apply. The operator function prototype and actual function definition for an overloaded operator has the following format (figure 7-1):

Figure 7-1: Syntax for Defining an Operator Overloading Function

```
<ReturnType> operator<OperatorSymbol> (<Parameter_list>);    // prototype
//…
<ReturnType><ClassName> :: operator <OperatorSymbol> (Parameter_list)
{
        // Body of operator function
        // ...
}
```

Here are a few clarifying related points:

1. The scope resolution operator (**::**) is required as shown in the syntax. Its purpose is to connect the intended operator to the related class and its redefined operation.

2. Typically, the return type for the operator function is the same as the host class that it appears in, but this needn't be the case.

3. In this context, **operator** is a keyword. Of course, this means that you are not allowed to use this keyword for any other purpose.

7.2 Overloading Binary Operators

As you are aware (from elementary algebra and earlier discussion of operators), a binary operator requires two operands for operation. Examples of binary operators in C++ include the following (review lecture 3 for the full list of operators):

```
+    -    *    /    &&   ==   ||    >=   <=   <    >    +=   -=   *=   /=   %
```

Some of these operators are arithmetic while others are relational. We will consider the arithmetic operators here.

7.2 Overloading Binary Operators (continued)

When a binary operator is overloaded, only the second operand needs to be sent as argument to the operator function. The first operand is implicit, being pointed to by the **this** pointer (review section 6.13). By convention, members pointed to by the **this** pointer are referenced without explicitly specifying the **this** pointer. Thus,

this→memberName is simply referred to as memberName

To further illustrate, suppose that we desire to store details about the sale activity of sales representatives for different commodities e.g. make of cars. Assume for simplicity, four makes of cars — Honda, Toyota, Chrysler, Volvo. For each sales person, we want to store the number of units sold and the sale amount for each make. We also want to be able to combine the information on a sales representative for analysis. For further analysis, we also want to be able to combine information for different sales representatives.

For instance, consider the following construct:

```
totalSales {Honda + Toyota + Chrysler + Volvo}
 = salesRepX {Honda + Toyota + Chrysler + Volvo} + salesRepY {Honda + Toyota + Chrysler + Volvo}
```

This construct could mean, add corresponding values for **salesRepX** and **salesRepY** to yield corresponding values for **totalSales**.

To develop a C++ specification for this scenario, we will define two classes — **SalesC** and **SalesRepC**. The former contains data members relating to actual sales of a salesman (object); the latter contains data members relating to personal information about each sales representative. Both classes will share a mutual friend function, **salesRepProfile**. The **SalesC** class will have overloaded operators for **+** and **=**.

Figure 7-1 shows the UML diagram for the **SalesRepC** class and the **SalesC** class. Figure 7-2 below is a representation of the scenario. Note how the overloaded operators are defined and used. Finally, figure 7-4 provides a more complete program listing of the problem, in the form of a program prototype for the management of an auto-store. Notice how the + and = operators have been overloaded to simplify the determination of total sales for the store.

Caution: When overloading non-commutative binary operators, it is important to remember which operand is on the left and which is on the right (e.g. A-B is not the same as B-A).

Figure 7-2a: UML Diagram for the SalesmanC Class

SalesRepC
protected int srNumber protected string srLastName, srFirstName
SalesRepC() void modify(SalesmanC, thisSalesman) void inputData(int x) string printMe() int getNumber() string getName()

Figure 7-2b: UML Diagram for the SalesC Class

SalesC
protected int saleSRNumber, salePeriod protected int saleHondaUnits, saleToyotaUnits, saleChryslerUnits, saleVolvoUnits protected double saleHondaAmount, saleToyotaAmount, saleChryslerAmount, saleVolvoAmount
SalesC() void modify(SalesmanC, thisSalesman) void inputData(int x) string printMe() SalesC operator+(SalesC sale2) SalesC operator=(SalesC sale2)

Figure 7-3: Overloading the + and the = Operators

```
class SalesRepC;      // forward declaration
class SalesC
{
  protected:
  int saleSRNumber, salePeriod, saleHondaUnits, saleToyotaUnits, saleChryslerUnits, saleVolvoUnits;
  float saleHondaAmount, saleToyotaAmount, saleChryslerAmount, saleVolvoAmount;

  public:
  // Member functions
  SalesC ( )  // Constructor defined as an in-line function
  {
   saleHondaUnits = saleToyotaUnits = saleChryslerUnits = saleVolvoUnits = 0;
   saleHonda Amount = saleToyotaAmount = saleChryslerAmount = saleVolvoAmount = 0;
  }
  void modify (SalesC thisSale);       // prototype
  SalesC operator + (SalesC sale2); // prototype; Sale1 is implied
  SalesC operator = (SalesC sale2); // Sale1 is implied
  friend void salesRepProfile (SalesRepC sRep, SalesC thisSale);
};

class SalesRepC
{
  protected:
  int srNumber;
  string srLastName, srFirstName;

  public:
  //member function prototypes
  SalesRepC ( );
  void modify (SalesRepC thisOne);
  friend void SalesRepProfile (SalesRepC sRep, SalesC thisSale);
};

// Member functions
void SalesC :: modify (SalesC thisSale)
{
   saleHondaUnits = thisSale.saleHondaUnits;
    // … similarly for other data members
  saleVolvoAmount = thisSale.saleVolvoAmount;
}

// overload +
SalesC SalesC :: operator + (SalesC sale2)
{
  // Note: this→HondaUnits is equivalent to HondaUnits, and so on…
  SalesC result;
  result.saleHondaUnits = saleHondaUnits + sale2.saleHondaUnits;
  result.saleToyotaUnits = saleToyotaUnits + sale2.saleToyotaUnits;
  // … similarly for other data members
  result.saleVolvoAmount = saleVolvoAmount + sale2.saleVolvoAmount;
  return result;
}
```

Figure 7-3: Overloading the + and the = Operators (continued)

```
// Overload operator =
SalesC SalesC:: operator = (SalesC Sale2)
{
   saleHondaUnits = sale2.saleHondaUnits;
    … similarly for other data members
   saleVolvoAmount = sale2.saleVolvoAmount;
   return *this;
}

// SalesRepC constructor
SalesRepC :: SalesRepC ( )
{
  int x;
  srNumber = 0; srFirstName = srLastName = " ";
}

// SalesRepC modifier
void SalesRepC :: modify (SalesRepC thisOne)
{
  srNumber = thisOne.srNumber;
  srFirstName = thisOne.srFirstName;
  srLastName = thisOne.srLastName;
}
// …
// The rest of the program file
int main(int argc, char *argv[])
{
   //…
   int sLim;
   SalesRepC salesRepList [sLim], curPerson;
   SalesC saleList[sLim], curSale, totalSales; curSale = SalesC(); totalSales = SalesC();
   //…
   for (int x= 1; x<=sLim; x++)
   {
          cin >> curPerson.srNumber; getchar();
          cin >> curPerson.srFirstName;  cin >> curPerson.srLastName;
          // …
          cin >> curSale.saleSRNumber; getchar();
          cin >> curSale.saleHondaUnits; getchar();
          // … similarly for other data members
          cin >> curSale.saleVolvoAmount; getchar();
          saleList[x - 1].modify(curSale);
          salesRepList[x-1].modify(curPerson);
          totalSales = totalSales + saleList[x - 1];  // cumulate the total sales via overloaded operators
          }
          //…
}         // End main

void salesRepProfile (SaleRepClass sRep, SalesC thisSale)
{
          // …    Has access to both SalesC and SalesRepC
}
```

Figure 7-4: The Auto-shop Program Prototype

```
// ****************************************************************************
// Program: Autoshop01 — to demonstrate operator overloading
// Author:  E. Foster
// ****************************************************************************
#include <cstdlib>
#include <iostream>
#include<ctype.h>
#include<string.h>
using namespace std;
// ****************************************************************************
// Class Definitions follow
// typedef char* cString;
class SalesRepC; // forward declaration

// The SalesC class
class SalesC
{
  protected:
  int saleSRNumber, salePeriod, saleHondaUnits, saleToyotaUnits, saleChryslerUnits, saleVolvoUnits;
  double saleHondaAmount, saleToyotaAmount, saleChryslerAmount, saleVolvoAmount;

  static const double HONDA_PRICE = 30000.00;
  static const double TOYOTA_PRICE = 33000.00;
  static const double CHRYSLER_PRICE = 32000.00;
  static const double VOLVO_PRICE = 35000.00;

  // Member functions
  public:
  SalesC ( )  // Constructor defined as an in-line function
  {
   saleSRNumber = salePeriod = saleHondaUnits = saleToyotaUnits = saleChryslerUnits = saleVolvoUnits = 0;
   saleHondaAmount = saleToyotaAmount = saleChryslerAmount = saleVolvoAmount = 0.00;
  }
  // Prototypes of other member functions
  void modify (SalesC thisSale);
  void inputData(int x);
  void printMe();
  SalesC operator+ (SalesC sale2);       // sale1 is implied
  SalesC operator= (SalesC sale2);       // sale1 is implied
  friend void salesmanProfile (SalesRepC thisSalesRep, SalesC thisSale);
};

// Continued on next  page
```

Figure 7-4: The Auto-shop Program Prototype (continued)

```
class SalesRepC
{
  protected:
  int srNumber;
  string srLastName; srFirstName;

  // member function prototypes
  public:
  SalesRepC(); // Constructor
  void modify (SalesRepC thisSalesman);
  void inputData(int x);
  void printMe();
  int getNumber();
  String getName();
  friend void salesmanProfile(SalesRepC thisSalesman, SalesC thisSale);
}; // End of Student class declaration

// The Member Functions of SalesC
// Constructor already defined as an inline function

// Member function Modify of SalesC
void SalesC :: modify (SalesC thisSale)
{
 saleSRNumber = thisSale.saleSRNumber;
 salePeriod = thisSale.salePeriod;
 saleHondaUnits = thisSale. saleHondaUnits; saleHondaAmount = thisSale. saleHondaAmount;
 saleToyotaUnits = thisSale. saleToyotaUnits; saleToyotaAmount = thisSale. saleToyotaAmount;
 saleChryslerUnits = thisSale. saleChryslerUnits; saleChryslerAmount = thisSale. saleChryslerAmount;
 saleVolvoUnits = thisSale. saleVolvoUnits; saleVolvoAmount = thisSale. saleVolvoAmount;
}

// Member function inputData of SalesC
void SalesC :: inputData (int x)
{
 // Accept units sold
 cout << "\nSale Information Entry for sales log# " << x << "\n\n";
 cout << "Please enter the following: \n";
 cout << "Salesman Number:   " ; cin >> saleSRNumber; getchar();
 cout << "\nSale Period (YYYYMM):  "; cin >> salePeriod; getchar();
 cout << "\nHonda Units:  "; cin >> saleHondaUnits; getchar();
 cout << "\nToyota Units:  "; cin >> saleToyotaUnits; getchar();
 cout << "\nChrysler Units:  "; cin >> saleChryslerUnits; getchar();
 cout << "\nVolvo Units:  "; cin >> saleVolvoUnits; getchar();

 // Calculate sale amounts
 saleHondaAmount = HsaleondaUnits * HONDA_PRICE;
 saleToyotaAmount = saleToyotaUnits * TOYOTA_PRICE;
 saleChryslerAmount = saleChryslerUnits * CHRYSLER_PRICE;
 saleVolvoAmount = saleVolvoUnits * VOLVO_PRICE;
}

// Continued on next page
```

Figure 7-4: The Auto-shop Program Prototype (continued)

```
// Member function printMe of SalesC
void SalesC :: printMe ( )
{
 // cout<< "\nSales Information: \n";
 cout<< "Salesman Number:   " << saleSRNumber << "\t" << "Sale Period: " << salePeriod << endl;
 cout<< "Honda Units:  " << HondaUnits << "\tHonda Amount: " << saleHondaAmount << endl;
 cout<< "Toyota Units:  " << ToyotaUnits << "\tToyota Amount: " << saleToyotaAmount << endl;
 cout<< "Chrysler Units:  " << ChryslerUnits << "\tChrysler Amount: " << saleChryslerAmount << endl;
 cout<< "Volvo Units:  " << VolvoUnits << "\tVolvo Amount: " << saleVolvoAmount << endl;
}

// Member function operator+ of SalesC - overloaded operator +
SalesC SalesC :: operator+ (SalesC sale2)
{
 SalesC result;
 result. saleHondaUnits = saleHondaUnits + sale2. saleHondaUnits;
 result. saleToyotaUnits = saleToyotaUnits + sale2. saleToyotaUnits;
 result. saleChryslerUnits = saleChryslerUnits + sale2. saleChryslerUnits;
 result. saleVolvoUnits = saleVolvoUnits + sale2. saleVolvoUnits;

 result. saleHondaAmount = saleHondaAmount + sale2. saleHondaAmount;
 result. saleToyotaAmount = saleToyotaAmount + sale2. saleToyotaAmount;
 result. saleChryslerAmount = saleChryslerAmount + sale2. saleChryslerAmount;
 result. saleVolvoAmount = saleVolvoAmount + sale2. saleVolvoAmount;
 return result;
}

// Member function operator= of SalesC - overloaded operator =
SalesC SalesC :: operator= (SalesC sale2)
{
 saleHondaUnits = sale2. saleHondaUnits; saleHondaAmount = sale2. saleHondaAmount;
 saleToyotaUnits = sale2. saleToyotaUnits; saleToyotaAmount = sale2. saleToyotaAmount;
 saleChryslerUnits = sale2. saleChryslerUnits; saleChryslerAmount = sale2. saleChryslerAmount;
 saleVolvoUnits = sale2. saleVolvoUnits; saleVolvoAmount = sale2. saleVolvoAmount;
 return *this;
}

// The Member Functions of SalesRepC
// Constructor of SalesRepC
SalesRepC :: SalesRepC( )
{
srNumber = 0;
srFirstName = srLastName = " ";
}

// Member function Modify of SalesRepC
void SalesRepC :: modify (SalesRepC thisSalesman)
{
 srNumber = thisSalesman.srNumber;
 srLastName = thisSalesman.srLastName;
 srFirstName = thisSalesman.srFirstName
}

// Continued on next page
```

Figure 7-4: The Auto-shop Program Prototype (continued)

```
// Member function inputData of SalesRepC
void SalesRepC :: inputData (int x)
{
 cout << "\nSalesman Information Entry for salesman " << x << "\n\n";
 cout << "Please enter the following: \n";
 cout << "Salesman Number:   " ; cin >>sr Number; getchar();
 cout << "\nSalesman Surname:  " ; gets(srLastName);
 cout << "\nSalesman First Name:  "; gets(srFirstName);
}

// Member function printMe of SalesRepC
void SalesRepC :: printMe ( )
{
 // cout<< "\nSalesman Information: \n";
 cout<< "Salesman Number:   " << srNumber << endl;
 cout<< "Salesman Name:  " << srFirstName << " " << srLastName << endl << endl;
}

// Member function getNumber of SalesRepC
int SalesRepC :: getNumber ( )
{ return srNumber; }

// Member function getName of SalesRepC
string SalesRepC :: getName ( )
{ string myName = " "; // String MyName = new char[31];  for (int x=1; x <= 30; x++) MyName[x-1] = ' ';
  myName srFirstName + " " + srLastName; // strcpy(myName, srFirstName);   strcat(myName, " ");   strcat(myName, srLastName);
  return myName;
}
// ***************************************************************************

// The rest of the program follows

// Global Variables and function prototypes
const String HEADING = "Autoshop Prototype";
void printList (SalesRepC srList[], int srLimit); // function prototype
void salesmanProfile(SalesRepC thisSalesRep, SalesC thisSale);

// Function printList
void printList (SalesRepC srList[], int srLimit)
{
 int x;
 cout << "\nYou have entered information for " << srLimit << " " << "Sales reps";
 cout << " as follows: \n";
 for (x = 1; x <= srLimit; x++) srList[x-1].printMe( );
} // End of PrintList function

// Function SalesmanProfile
void salesmanProfile(SalesRepC thisSalesman, SalesC thisSale)
{
    // ...
}
// Continued on next page
```

Figure 7-4: The Auto-shop Program Prototype (continued)

```
// Main function
int main(int argc, char *argv[])
{
 // Declarations
 bool exitTime = false;   char exitKey;          int x, srLim, sLim;       SalesC totalSales;

 while (!exitTime) // While user wishes to continue
 {
  // Prompt the user for the number of sales reps objects and create the array
  cout << HEADING;

  // Prompt for information on each salesman
  cout << "\n\nPlease enter the number of sales reps required: ";
  cin >> srLim; getchar();
  SalesmanC salesRepList[srLim];              // Array of srLim SalesmanC objects
  for (x = 1; x <=srLim; x++)
  {       salesRepList[x-1] = SalesmanC();    // Instantiate the item
          salesRepList[x-1].inputData(x);      // Obtain  information for the item
  }

  // Print the list
  printList(salesRepList, srLim);

  // Prompt for information on sales
  cout << "\n\nPlease enter the number of sales logs to be entered: ";
  cin >> sLim; getchar();
  SalesC saleList[sLim];          // Array of sLim SalesC objects
  totalSales = SalesC(); // Instantiate TotalSales
  for (x = 1; x <=sLim; x++)
  {       saleList [x-1] = SalesC(); // Instantiate the item
          saleList [x-1].inputData(x);   // Obtain  information for the item
          totalSales = totalSales + saleList [x-1]; // cumulate totalSales via the overloaded operators
  }

  // Print Total Sales
  cout << "\nTotal Sales Entered: " << endl;
  totalSales.printMe();

  // Find out whether user wishes to continue
  cout << "\n Press any key to continue or X to exit ";
  exitKey = getchar();
  if (toupper(exitKey) == 'X') exitTime = true;

 } // End of While-user -wishes-to-continue

 system("PAUSE");
 return EXIT_SUCCESS;
} // End of main function
```

In observing the code in figure 7-4, note that the classes **SalesRepC** and **SalesC** can be extracted from the program file and placed in separate header files. In doing so, you would end up with two header files (one per class) and a third file (which could also be defined as a driver class) containing the **main**(. . .) function and other related functions. This is the recommended approach.

7.3 Overloading Unary Operators

Unary operators may also be overloaded. Examples of unary operators include (review lecture 3):

++ -- - +

For unary operations, no object is passed to the operator; since the **this** pointer points to the object making the call, the operand is implicitly known to the function.

Sticking with the auto store example, we may overload the increment operator (++) to increment the number of units sold for each car model. Figure 7-5 illustrates how this may be accomplished.

Figure 7-5: Overloading the ++ Operator:

```
// Assuming the declarations of example 1, define an overloaded prefix ++ operation

class SalesRepC; // Forward declaration
class SalesC {
protected:
int: saleHondaUnits, saleToyotaUnits, saleChryslerUnits, saleVolvoUnits;
double saleHondaAmount, saleToyotaAmount, saleChryslerAmount, saleVolvoAmount;

// Member functions
public:
SalesC ( ) // Constructor defined as an in-line function
{
  saleHondaUnits = saleToyotaUnits = saleChryslerUnits = saleVolvoUnits = 0;
  saleHonda Amount = saleToyotaAmount = saleChryslerAmount = saleVolvoAmount = 0.0;
}
void modify (SalesC, ThisSale);       // prototype
SalesC operator + (SalesC Sale2);  // prototype; Sale1 is implied
SalesC operator = (SalesC Sale2);  // Sale1 is implied
SalesC operator ++ ( );                  // prefix ++
SalesC operator ++ (int NotUsed);   // postfix ++
friend void salesmanProfile (SalesRepC thisRep, SalesC thisSale);
};

class SalesmanC {
// … // as in Example figure 7-4
};
… // Member functions as in figure 7-4
SalesC SalesC :: operator++ ( )       // prefix version of ++
{
  saleHondaUnits++; saleToyotaUnits++; saleChryslerUnits++; saleVolvoUnits++;
  saleHondaAmount =  saleHondaUnits * HONDA_PRICE;
  saleToyotaAmount = saleToyotaUnits * TOYOTA_PRICE;
  saleChryslerAmount = saleChryslerUnits * CHRYSLER_PRICE;
  saleVolvoAmount = saleVolvoUnits * VOLVO_PRICE;
  return *this;
}
```

Figure 7-5: Overloading the ++ Operator (continued):

```
SalesC SalesC :: operator++ (int notUsed)          // postfix version
{
  SalesC temp = *this          // saves before incrementing
  saleHondaUnits++; saleToyotaUnits++; saleChryslerUnits++; saleVolvoUnits++;
  saleHondaAmount =  saleHondaUnits * HONDA_PRICE;
  saleToyotaAmount = saleToyotaUnits * TOYOTA_PRICE;
  saleChryslerAmount = saleChryslerUnits * CHRYSLER_PRICE;
  saleVolvoAmount = saleVolvoUnits * VOLVO_PRICE;
  return temp;// returns original value
}
// …
int main(int argc, char *argv[])
{ ..      // as before in figure 7-4
++totalSales;          // increments before use
// …
totalSales++;          // increments after use
…
}
```

As you observe the sample code (of figure 7-5), here are a few points of clarification about overloading the increment/decrement operator:

1. In the example, there two are overloaded operator ++ functions: one for prefix and the other for postfix. The postfix form includes an unused parameter. It is there simply to indicate to C++ that the unary operator is in postfix form.

2. Older C++ compilers may not support both prefix and postfix unary operator functions, so beware.

3. In the interest of clarity, try to use overloaded operators to reflect the intent of the operator's original use (e.g. + relates to addition; do not overload it for non-additive activities).

4. You cannot alter the precedence of an operator, or the number of operands it requires. Your overloaded operator function must assume the originally number of operands for the operator.

5. Except for function call operators, operator functions cannot have default arguments.

7.4 Friend Operator Functions

It is possible to define an operator overload function as a friend function instead of a member function. This strategy is useful for the scenario where the overloaded operator is to apply to more than one class. Rather than defining it in several classes, it is more prudent to define a set of overloaded friend functions — one for each related class. All classes that need to use it can then bestow friendship to this function.

7.4 Friend Operator Functions (continued)

Note: If the operator overload function is a friend function, it does not have the facility of the **this** pointer at its disposal. You must therefore explicitly declare all the operands required (one for unary operator, two for binary operator, etc) as parameters. You must also declare an object of the return type (typically a host class) for the purpose of returning a value. Still using the auto store as frame of reference, figure 7-6 provides an example.

Figure 7-6: Implementing the Overloaded Operator as a Friend Function

```
class SalesRepC;
class SalesC
{
…  // as in figure 7-4
friend SalesC operator+ (SalesC sale1, SalesC sale2);
…
};

class SalesRepC
{
…       // as in figure 7-4
};
          …
int main(int argc, char *argv[])
{
 … // as in figure 7-4
}

// Functions
// …
SalesC operator + (SalesC sale1, sale2)
{
  SalesC result;
  result.saleHondaUnits = sale1.saleHondaUnits + sale2.saleHondaUnits;
  result.saleToyotaUnits = sale1.saleToyotaUnits + sale2.saleToyotaUnits;
  result.saleChrslerUnits = sale1.saleChryslerUnits + sale2.saleChryslerUnits;
  result.saleVolvoUnits = sale1.saleVolvoUnits + sale2.saleVolvoUnits;

  result.saleHondaAmount = sale1.saleHondaAmount + sale2.saleHondaAmount;
  result.saleHondaAmount = sale1.saleHondaAmount + sale2.saleHondaAmount;
  result.saleHondaAmount = sale1.saleHondaAmount + sale2.saleHondaAmount;
  result.saleHondaAmount = sale1.saleHondaAmount + sale2.saleHondaAmount;

  return result;
}
```

Note: For unary operators, the friend function does not have the **this** pointer. To access the called operand, you must declare the parameter as a pointer to the object; or better yet, a reference parameter.

7.5 Overloading Relational Operators

Relational operators (e.g. < > <= >= ==) may be overloaded in a manner similar to other operators, but with one distinction: relational operators return the value **true** or **false**. Continuing with the auto-store example, figure 7-7 provides an example. The assumed convention is that a **SalesC** object **sale1** is greater than another **Sales**C object **sale2** if all each component of **sale1** is greater than its corresponding component in **sale2**.

Figure 7-7: Overloading the > Operator

```
class SalesRepC;
class SalesC
{
  protected:
  int: saleHondaUnits, saleToyotaUnits, saleChryslerUnits, saleVolvoUnits;
  double saleHondaAmount, saleToyotaAmount, saleChryslerAmount, saleVolvoAmount;

  // Member functions
  public:
  SalesC ( )
  { saleHondaUnits = saleToyotaUnits = saleChryslerUnits = saleVolvoUnits = 0;
    saleHonda Amount = saleToyotaAmount = saleChryslerAmount = saleVolvoAmount = 0.0;
  }

  void modify (SalesC, ThisSale);     // prototype
  SalesC operator + (SalesC Sale2);// prototype; Sale1 is implied
  SalesC operator = (SalesC Sale2);// Sale1 is implied
  friend void salesmanProfile (SalesmanC Sman, SalesC ThisSale);
  bool operator > (SalesC Sale2);
};

class SalesRepC
{
 // …    as in figure 7-4
}
// … as in figure 7-4

bool SalesC :: operator > (SalesC sale2)
{
   if ((saleHondaUnits > sale2.saleHondaUnits) &&…. && (saleVolvoAmount > sale2.saleVolvoAmount))
   return true;
   else return false;
}

int main(int argc, char *argv[])
{
SalesC Bruce, Joan;
//…
if (Bruce > Joan) cout<< "Well done Bruce";
 // …
}
```

7.6 Summary and Concluding Remarks

Here is the summary of what we have covered in this lecture:

- Operator overloading is the act of redefining the behavior of an operation. It is one ways that C++ facilitates polymorphism.
- If an operator is to be overloaded for instances of a class, a special operator function must be defined within the class. This function specifies the new behavior of the operator.
- When a binary operator is overloaded within a class, only the second operand needs to be defined as a parameter to the operator function; the first operand is implied as it is accessible via the **this** pointer.
- When a unary operator is overloaded, no operand needs to be passed to the operator function; the operand is accessible via the **this** pointer.
- Relational operators are also binary operators and therefore may be overloaded. The guideline for binary operators also applies to relational operators.
- Operator functions may be defined as friend functions. In this case, the guidelines for operands do not apply: the operands must be explicitly defined as parameters.

Other operators such as […] and (…) may be overloaded, according to the situation. The only C++ operators that cannot be overloaded are the following:

- Member access or dot operator (→ or .)
- Ternary or conditional operator (?:)
- Scope resolution operator (::)
- Pointer-to-member operator (.*)
- Object size operator (**sizeof**)
- Object type operator (**typeid**)
- Memory allocation operator (**new**)
- Memory deallocation operator (**delete**)

In overloading operators, take care not to return an object, or pass an object as argument to an operator overload function, if the object's destructor de-allocates memory space that was allocated for the object.

Operator overloading is one of the distinguishing features of the C++ language. It provides the programmer with flexibility and the power to write very complex programs that are lean and efficient. Many programming languages do not support this feature, while others support it minimally.

The next lecture discusses inheritance.

7.7 Recommended Readings

[Savitch 2013] Savitch, Walter. 2013. *Absolute C++*, 5th Edition. Boston: Pearson. See chapter 8.

[Savitch 2015] Savitch, Walter. 2015. *Problem Solving with C++*, 9h Edition. Boston: Pearson. See chapter 11.

[Yang 2001] Yang, Daoqi. 2001. *C++ and Object-Oriented Numeric Computing for Scientists and Engineers*. New York, NY: Springer. See chapter 6.