Fundamentals of Operator Overloading

• Use operators with objects (operator overloading)
  – Clearer than function calls for certain classes
  – Operator sensitive to context

• Types
  – Built in (int, char) or user-defined
  – Can use existing operators with user-defined types
    • Cannot create new operators

• Overloading operators
  – Create a function for the class
  – Name function operator followed by symbol
    • Operator+ for the addition operator +
Fundamentals of Operator Overloading

• Using operators on a class object
  – It must be overloaded for that class
  – Exceptions:
    • Assignment operator, =
      – May be used without explicit overloading
      – Memberwise assignment between objects
    • Address operator, &
      – May be used on any class without overloading
      – Returns address of object
  • Both can be overloaded
Restrictions on Operator Overloading

• Cannot change
  – How operators act on built-in data types
    • I.e., cannot change integer addition
  – Precedence of operator (order of evaluation)
    • Use parentheses to force order-of-operations
  – Associativity (left-to-right or right-to-left)
  – Number of operands
    • & is unitary, only acts on one operand

• Cannot create new operators

• Operators must be overloaded explicitly
  – Overloading + does not overload +=
Restrictions on Operator Overloading

<table>
<thead>
<tr>
<th>Operators that can be overloaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
</tr>
<tr>
<td>~</td>
</tr>
<tr>
<td>/=</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
</tr>
<tr>
<td>--</td>
</tr>
<tr>
<td>new[]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operators that cannot be overloaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
</tr>
</tbody>
</table>
Operator Functions As Class Members Vs. As Friend Functions

• Operator functions
  – Member functions
    • Use this keyword to implicitly get argument
    • Gets left operand for binary operators (like +)
    • Leftmost object must be of same class as operator
  – Non member functions
    • Need parameters for both operands
    • Can have object of different class than operator
    • Must be a friend to access private or protected data

• Example Overloaded \(<\!\!<\) operator
  – Left operand of type ostream &
    • Such as cout object in cout \(<\!\!<\) classObject
  – Similarly, overloaded \(\rangle\!\rangle\) needs istream &
  – Thus, both must be non-member functions
Operator Functions As Class Members Vs. As Friend Functions

• Commutative operators
  – May want + to be commutative
    • So both “a + b” and “b + a” work
  – Suppose we have two different classes
  – Overloaded operator can only be member function when its class is on left
    • HugeIntClass + Long int
    • Can be member function
  – When other way, need a non-member overload function
    • Long int + HugeIntClass
Overloading Stream-Insertion and Stream-Extraction Operators

• `<<` and `>>`
  - Already overloaded to process each built-in type
  - Can also process a user-defined class

• Example program
  - Class `PhoneNumber`
    • Holds a telephone number
  - Print out formatted number automatically
    • `(123) 456–7890`
// Fig. 8.3: fig08_03.cpp
// Overloading the stream-insertion and
// stream-extraction operators.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;
using std::ostream;
using std::istream;

#include <iomanip>

using std::setw;

// PhoneNumber class definition
class PhoneNumber {

friend ostream &operator<<( ostream&, const PhoneNumber & );
friend istream &operator>>( istream&, PhoneNumber & );

private:
    char areaCode[4]; // 3-digit area code and null
    char exchange[4]; // 3-digit exchange and null
    char line[5]; // 4-digit line and null

}; // end class PhoneNumber

Notice function prototypes for
overloaded operators >> and <<
They must be non-member friend
functions, since the object of class
PhoneNumber appears on the right of
the operator.

cin << object
cout >> object
// overloaded stream-insertion operator; cannot be a member function if we would like to invoke it with cout << somePhoneNumber;

ostream &operator<<( ostream &output, const PhoneNumber &num )
{
    output << "(" << num.areaCode << "") "
    << num.exchange << "-" << num.line;

    return output; // enables cout << a << b << c;
}

// overloaded stream-extraction operator; cannot be a member function if we would like to invoke it with cin >> somePhoneNumber;

istream &operator>>( istream &input, PhoneNumber &num )
{
    input.ignore();                     // skip ( (4) >> num.areaCode; // input area code
    input.ignore( 2 );                  // skip ) and space
    input >> setw( 4 ) >> num.exchange; // input exchange
    input.ignore();                    // skip dash (-)
    input >> setw( 5 ) >> num.line;     // input line

    return input;      // enables cin >> a >> b >> c;

    The expression:
    cout << phone;
    is interpreted as the function call:
    operator<<(cout, phone);

    output is an alias for cout.

    This allows objects to be cascaded.
    cout << phone1 << phone2;
    first calls
    operator<<(cout, phone1), and
    returns cout.

    Next, cout << phone2 executes.

    Stream manipulator setw restricts number of characters read. setw(4) allows 3 characters to be read, leaving room for the null character.
int main()
{
    PhoneNumber phone; // create object phone
    cout << "Enter phone number in the form (123) 456-7890:\n";
    cin >> phone; // cin >> phone invokes operator>> by implicitly issuing
    // the non-member function call operator>>( cin, phone )
    cout << "The phone number entered was: " ;
    cout << phone << endl;
    return 0;
} // end main

Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
Overloading Unary Operators

• Overloading unary operators
  – Non-\texttt{static} member function, no arguments
  – Non-member function, one argument
    • Argument must be class object or reference to class object
  – Remember, \texttt{static} functions only access \texttt{static} data
Overloading Operators

• Overloading unary operators (\(!\) to test for empty string)
  – Non-\textit{static} member function: \texttt{!s} becomes \texttt{s.operator!()}\newline    \texttt{bool operator!() const;}
  – Non-member function: \texttt{s!} becomes \texttt{operator!(s)}\newline    \texttt{friend bool operator!( const String & );}

• Overloading binary operators
  – Non-member function (arg. must be class object or reference)\newline    \texttt{friend const String &operator+=(String &, const String & );}
  – Non-\textit{static} member function:\newline    \texttt{const String &operator+=( const String & );}
  – \texttt{y += z} equivalent to \texttt{y.operator+=( z )}
Case Study: Array class

• Arrays in C++
  – No range checking
  – Cannot be compared meaningfully with ==
  – No array assignment (array names const pointers)
  – Cannot input/output entire arrays at once

• Example: Implement an Array class with
  – Range checking
  – Array assignment
  – Arrays that know their size
  – Outputting/inputting entire arrays with << and >>
  – Array comparisons with == and !=
Case Study: Array class

• Copy constructor
  – Used whenever copy of object needed
    • Passing by value (return value or parameter)
    • Initializing an object with a copy of another
      – `Array newArray( oldArray );`
      – `newArray` copy of `oldArray`
  – Prototype for class **Array**
    • `Array( const Array & );`
    • *Must* take reference
      – Otherwise, pass by value
      – Tries to make copy by calling copy constructor…
      – Infinite loop
// Fig. 8.4: array1.h
// Array class for storing arrays of integers.
#ifndef ARRAY1_H
#define ARRAY1_H

#include <iostream>

using std::ostream;
using std::istream;

class Array {
    friend ostream &operator<<( ostream &, const Array & );
    friend istream &operator>>( istream &, Array & );

public:
    Array( int = 10 ); // default constructor
    Array( const Array & ); // copy constructor
    ~Array(); // destructor
    int getSize() const; // return size

    // assignment operator
    const Array &operator=( const Array & );

    // equality operator
    bool operator==( const Array & ) const;
};

Most operators overloaded as member functions (except << and >>, which must be non-member functions).

Prototype for copy constructor.
// inequality operator; returns opposite of == operator
bool operator!=(const Array &right) const
{
    return !(*this == right); // invokes Array::operator==
}

// subscript operator for non-const objects returns lvalue
int &operator[](int);

// subscript operator for const objects returns rvalue
const int &operator[](int) const;

private:
    int size; // array size
    int *ptr; // pointer to first element of array
}; // end class Array

#endif

!= operator simply returns opposite of == operator. Thus, only need to define the == operator.
// Fig 8.5: array1.cpp
// Member function definitions for class Array
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <iomanip>

using std::setw;

#include <new>      // C++ standard "new" operator
#include <cstdlib>   // exit function prototype
#include "array1.h" // Array class definition

// default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
    // validate arraySize
    size = ( arraySize > 0 ? arraySize : 10 );
    ptr = new int[ size ]; // create space for array
```cpp
for ( int i = 0; i < size; i++ )
    ptr[ i ] = 0;          // initialize array
}

// copy constructor for class Array;
// must receive a reference to prevent infinite recursion
Array::Array( const Array &arrayToCopy )
    : size( arrayToCopy.size )
{
    ptr = new int[ size ]; // create space for array
    for ( int i = 0; i < size; i++ )
        ptr[ i ] = arrayToCopy.ptr[ i ];  // copy into object
}

// destructor for class Array
Array::~Array()
{
    delete [] ptr;  // reclaim array space
}
```

We must declare a new integer array so the objects do not point to the same memory.
51 // return size of array
52 int Array::getSize() const
53 {
54     return size;
55 }
56 } // end function getSize
57
58 // overloaded assignment operator;
59 // const return avoids: (a1 = a2) = a3
60 const Array &Array::operator=( const Array &right )
61 {
62     if ( &right != this ) {  // check for self-assignment
63         // for arrays of different sizes, deallocate original
64         // left-side array, then allocate new left-side array
65         if ( size != right.size ) {
66             delete [] ptr;       // reclaim space
67             size = right.size;  // resize this object
68             ptr = new int[size]; // create space for array copy
69         }
70     }
71 } // end inner if
72
73     for ( int i = 0; i < size; i++ )
74         ptr[i] = right.ptr[i]; // copy array into object
75 } // end outer if

Want to avoid self-assignment.
return *this; // enables x = y = z, for example
}

// determine if two arrays are equal and
// return true, otherwise return false
bool Array::operator==( const Array &right ) const
{
    if ( size != right.size )
        return false;    // arrays of different sizes
    for ( int i = 0; i < size; i++ )
        if ( ptr[ i ] != right.ptr[ i ] )
            return false; // arrays are not equal
    return true;       // arrays are equal
}

} // end function operator==
// overloaded subscript operator for non-const Arrays
int &Array::operator[]( int subscript )
{
    // check for subscript out of range error
    if ( subscript < 0 || subscript >= size ) {
        cout << "\nError: Subscript " << subscript
            << " out of range" << endl;
        exit( 1 );  // terminate program; subscript out of range
    }
    return ptr[ subscript ]; // reference return
} // end function operator[]

integers1[5] calls integers1.operator[]( 5 )

exit() (header <cstdlib>) ends the program.
115 // overloaded subscript operator for const Arrays
116 // const reference return creates an rvalue
117 const int &Array::operator[]( int subscript ) const
118 {
119     // check for subscript out of range error
120     if ( subscript < 0 || subscript >= size ) {
121         cout << "Error: Subscript " << subscript
122             << " out of range" << endl;
123         exit( 1 );  // terminate program; subscript out of range
124     } // end if
125     return ptr[ subscript ]; // const reference return
126 } // end function operator[]
127
128 // overloaded input operator for class Array;
129 // inputs values for entire array
130 istream &operator>>( istream &input, Array &a )
131 {
132     for ( int i = 0; i < a.size; i++ )
133         input >> a.ptr[ i ];
134     return input;   // enables cin >> x >> y;
135 } // end function
142 // overloaded output operator for class Array
143 ostream &operator<<( ostream &output, const Array &a )
144 {
145   int i;
146
147   // output private ptr-based array
148   for ( i = 0; i < a.size; i++ ) {
149     output << setw( 12 ) << a.ptr[ i ];
150     if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
151       output << endl;
152   }
153
154   // end for
155
156   if ( i % 4 != 0 ) // end last line of output
157     output << endl;
158
159   return output; // enables cout << x << y;
160
161 } // end function operator<<
Converting between Types

- **Cast operator (conversion operator)**
  - Convert from One class to another built-in type
  - Must be non-`static` member function -
    - Cannot be `friend`
  - Do not specify return type
    - Implicitly returns type to which you are converting
  - Example: `A::operator char *() const;`
    - Casts class `A` to a temporary `char *`
    - `(char *)s` calls `s.operator char*()`

- Casting can prevent need for overloading
  - Suppose class `String` can be cast to `char *`
  - `cout << s;` // `cout` expects `char *`; `s` is a `String`
    - Compiler implicitly calls the function to convert `s` to `char *`
    - Do not have to overload `<< for String`
Case Study: A String Class

• Build class String
  – String creation, manipulation
  – Class string in standard library (more Chapter 15)

• Conversion constructor
  – Single-argument constructor
  – Turns objects of other types into class objects
    • String s1(“hi”);
    • Creates a String from a char *
  – Any single-argument constructor is a conversion constructor
Overloading ++ and --

- Increment/decrement operators can be overloaded
  - Add 1 to a Date object, d1
  - Prototype (member function)
    - Date &operator++();
    - ++d1 same as d1.operator++()
  - Prototype (non-member)
    - Friend Date &operator++( Date &);
    - ++d1 same as operator++( d1 )
Overloading ++ and --

• To distinguish pre/post increment
  – Post increment has a dummy parameter
    • int of 0
  – Prototype (member function)
    • Date operator++( int );
    • d1++ same as d1.operator++( 0 )
  – Prototype (non-member)
    • friend Date operator++( Data &, int );
    • d1++ same as operator++( d1, 0 )
  – Integer parameter does not have a name
    • Not even in function definition
Overloading ++ and --

- Return values
  - Preincrement
    - Returns by reference (Date &)
    - lvalue (can be assigned)
  - Postincrement
    - Returns by value
    - Returns temporary object with old value
    - rvalue (cannot be on left side of assignment)

- Example Date class
  - Overloaded increment operator
    - Change day, month and year
  - Overloaded += operator
  - Function to test for leap years
  - Function to determine if day is last of month
// Fig. 8.10: date1.h
// Date class definition.
#ifndef DATE1_H
#define DATE1_H
#include <iostream>

using std::ostream;

class Date {
    friend ostream &operator<<( ostream &, const Date & );

public:
    Date( int m = 1, int d = 1, int y = 1900 ); // constructor
    void setDate( int, int, int ); // set the date

    Date &operator++();            // preincrement operator
    Date operator++( int );        // postincrement operator

    const Date &operator+=( int ); // add days, modify object

    bool leapYear( int ) const;   // is this a leap year?
    bool endOfMonth( int ) const; // is this end of month?
}

Note difference between pre and post increment.
23 private:
24 int month;
25 int day;
26 int year;
27
28 static const int days[];       // array of days per month
29 void helpIncrement();         // utility function
30
31 }; // end class Date
32
33 #endif

Date &Date::operator++()
36 {
37    helpIncrement();
38    return *this; // reference return to create an lvalue
39 } // end function operator++
40
41 // overloaded postincrement operator; note that the dummy
42 // integer parameter does not have a parameter name
43 Date Date::operator++( int )
44 {
45    Date temp = *this; // hold current state of object
46    helpIncrement();
47    // return unincremented, saved, temporary object
48    return temp;  // value return; not a reference return
49 } // end function operator++
Inheritance

- Inheritance
  - Software reusability
  - Create new class from existing class
    - Absorb existing class’s data and behaviors
    - Enhance with new capabilities
  - Derived class inherits from base class
    - Derived class
      - More specialized group of objects
      - Behaviors inherited from base class
        - Can customize
      - Additional behaviors
Inheritance

• Class hierarchy
  – Direct base class
    • Inherited explicitly (one level up hierarchy)
  – Indirect base class
    • Inherited two or more levels up hierarchy
  – Single inheritance
    • Inherits from one base class
  – Multiple inheritance
    • Inherits from multiple base classes
      – Base classes possibly unrelated
    • Chapter 22
Three types of inheritance

- **public**
  - Every object of derived class also object of base class
    - Base-class objects not objects of derived classes
    - Example: All cars vehicles, but not all vehicles cars
  - Can access non-**private** members of base class
    - Derived class can effect change to **private** base-class members
      - Through inherited non-**private** member functions

- **private**
  - Alternative to composition
  - Chapter 17

- **protected**
  - Rarely used
Inheritance

• Abstraction
  – Focus on commonalities among objects in system

• “is-a” vs. “has-a”
  – “is-a”
    • Inheritance
    • Derived class object treated as base class object
    • Example: Car is a vehicle
      – Vehicle properties/behaviors also car properties/behaviors
  – “has-a”
    • Composition
    • Object contains one or more objects of other classes as members
    • Example: Car has a steering wheel
Base Classes and Derived Classes

• Base classes and derived classes
  – Object of one class “is an” object of another class
    • Example: Rectangle is quadrilateral.
  – Base class typically represents larger set of objects than derived classes
    • Example:
      – Base class: Vehicle
        • Cars, trucks, boats, bicycles, …
      – Derived class: Car
        • Smaller, more-specific subset of vehicles
## Base Classes and Derived Classes

### Inheritance examples

<table>
<thead>
<tr>
<th>Base class</th>
<th>Derived classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
<td>GraduateStudent</td>
</tr>
<tr>
<td></td>
<td>UndergraduateStudent</td>
</tr>
<tr>
<td>Shape</td>
<td>Circle</td>
</tr>
<tr>
<td></td>
<td>Triangle</td>
</tr>
<tr>
<td></td>
<td>Rectangle</td>
</tr>
<tr>
<td>Loan</td>
<td>CarLoan</td>
</tr>
<tr>
<td></td>
<td>HomeImprovementLoan</td>
</tr>
<tr>
<td></td>
<td>MortgageLoan</td>
</tr>
<tr>
<td>Employee</td>
<td>FacultyMember</td>
</tr>
<tr>
<td></td>
<td>StaffMember</td>
</tr>
<tr>
<td>Account</td>
<td>CheckingAccount</td>
</tr>
<tr>
<td></td>
<td>SavingsAccount</td>
</tr>
</tbody>
</table>
Base Classes and Derived Classes

• Inheritance hierarchy
  – Inheritance relationships: tree-like hierarchy structure
  – Each class becomes
    • Base class
      – Supply data/behaviors to other classes
    OR
    • Derived class
      – Inherit data/behaviors from other classes
Inheritance hierarchy

Fig. 9.2 Inheritance hierarchy for university CommunityMembers.
Inheritance hierarchy

Fig. 9.3  Inheritance hierarchy for Shapes.
Base Classes and Derived Classes

• **public** inheritance
  
  - Specify with:
    
    ```
    Class TwoDimensionalShape : public Shape
    
    • Class TwoDimensionalShape inherits from class Shape
    ```
  
  - Base class **private** members
    
    • Not accessible directly
    
    • Still inherited - manipulate through inherited member functions
  
  - Base class **public** and **protected** members
    
    • Inherited with original member access
  
  - **friend** functions
    
    • Not inherited
protected Members

• protected access
  – Intermediate level of protection between public and private
  – protected members accessible to
    • Base class members
    • Base class friends
    • Derived class members
    • Derived class friends
  – Derived-class members
    • Refer to public and protected members of base class
      – Simply use member names
Relationship between Base Classes and Derived Classes

• Base class and derived class relationship
  – Example: Point/circle inheritance hierarchy
    • Point
      – x-y coordinate pair
    • Circle
      – x-y coordinate pair
      – Radius
Relationship between Base Classes and Derived Classes

• Using *protected* data members
  – Advantages
    • Derived classes can modify values directly
    • Slight increase in performance
      – Avoid set/get function call overhead
  – Disadvantages
    • No validity checking
      – Derived class can assign illegal value
    • Implementation dependent
      – Derived class member functions more likely dependent on base class implementation
      – Base class implementation changes may result in derived class modifications
        • Fragile (brittle) software
Case Study: Three-Level Inheritance Hierarchy

- Three level point/circle/cylinder hierarchy
  - Point
    - x-y coordinate pair
  - Circle
    - x-y coordinate pair
    - Radius
  - Cylinder
    - x-y coordinate pair
    - Radius
    - Height
Constructors and Destructors in Derived Classes

• Instantiating derived-class object
  – Chain of constructor calls
    • Derived-class constructor invokes base class constructor
      – Implicitly or explicitly
    • Base of inheritance hierarchy
      – Last constructor called in chain
      – First constructor body to finish executing
      – Example: \texttt{Point3/Circle4/Cylinder} hierarchy
        • \texttt{Point3} constructor called last
        • \texttt{Point3} constructor body finishes execution first
  • Initializing data members
    – Each base-class constructor initializes data members inherited by derived class
Constructors and Destructors in Derived Classes

• Destroying derived-class object
  – Chain of destructor calls
    • Reverse order of constructor chain
    • Destructor of derived-class called first
    • Destructor of next base class up hierarchy next
      – Continue up hierarchy until final base reached
        • After final base-class destructor, object removed from memory

• Base-class constructors, destructors, assignment operators
  – Not inherited by derived classes
  – Derived class constructors, assignment operators can call
    • Constructors
    • Assignment operators
<table>
<thead>
<tr>
<th>Base class member access specifier</th>
<th>Type of inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>public inheritance</td>
</tr>
<tr>
<td><strong>Public</strong></td>
<td><code>public</code> in derived class. Can be accessed directly by any non-static member functions, friend functions and non-member functions.</td>
</tr>
<tr>
<td><strong>Protected</strong></td>
<td><code>protected</code> in derived class. Can be accessed directly by all non-static member functions and friend functions.</td>
</tr>
<tr>
<td><strong>Private</strong></td>
<td>Hidden in derived class. Can be accessed by non-static member functions and friend functions through public or protected member functions of the base class.</td>
</tr>
</tbody>
</table>
Software Engineering with Inheritance

• Customizing existing software
  – Inherit from existing classes
    • Include additional members
    • Redefine base-class members
    • No direct access to base class’s source code
      – Link to object code
  – Independent software vendors (ISVs)
    • Develop proprietary code for sale/license
      – Available in object-code format
    • Users derive new classes
      – Without accessing ISV proprietary source code