CSE 230
Intermediate Programming in C and C++
Operator Overloading

Fall 2017
Stony Brook University
Instructor: Shebuti Rayana

http://www3.cs.stonybrook.edu/~cse230/

Ref. Book: C How to Program, 8th edition by Deitel and Deitel
Introduction

- How to enable C++’s operators to work with class objects—a process called operator overloading.

- The jobs performed by overloaded operators also can be performed by explicit function calls, but operator notation is often more natural.
Consider the following examples:

```cpp
date d;
d.increment();

bag b;
cout << b.getData(i);
b.setdata(i, value);

matrix x, y, z;
x.add(y);
multiply(x, y, z);
```
General View (cont.)

- How do you prefer the replacements below?

```cpp
Date d;
d.increment();         d++;  
Bag b;
cout << b.getData(i);  cout << b[i];
b.setData(i, value);   b[i] = value;

Matrix x, y, z;
x.add(y);              x += y;
multiply(x, y, z);     x = y * z;
```
Manipulation on class objects are accomplished by sending messages (by function calls) to the objects.

Function-call notation is cumbersome for certain kinds of classes, especially mathematical classes.

It would be nice to use C++’s rich set of built-in operators to specify object manipulation.

For example, operator << (or >>, +, -, etc.) has several purposes as the stream-insertion and bitwise left-shift.

Overloaded operators perform operation depending on their context and set of operands.
Fundamentals of Operator Overloading

- Programmers can define user-defined types and use operators with user-defined types.

- New operators can not be created but existing operators may be overloaded, so when they are used with class objects, they have meaning appropriate to the new types.

- This is one of C++’s most powerful features.

- The assignment (=) and address (&) operators may be used with objects of any class without overloading. But they can be overloaded also.

- Programmer must explicitly write operator overloading function to perform a desired operation. These functions may be defined as a member function, friend, etc.

- Extreme Misuse is not recommended! (* for addition, + for division, etc.)
Implementation Issues

■ When overloading ( ), [ ], -> or any other assignment operators, the operator overloading function must be declared as a class member.

■ For other operators, the operator overloaded function can be non-member functions.

■ When an operator function is a member function, the leftmost (or only) operand must be a class object (or a reference to a class object) of the operator’s class.

■ If the left operand is an object of a different class or a built-in type, this function must be a non-member.

■ The keyword `operator` followed by the operator replaces the name of a function, for example `className::operator = (...)`
Implementation Issues (cont.)

- A unary operator for a class can be overloaded as a class member function with no arguments or as a non-member function with one argument; that argument must be either an object of the class or a reference to an object of the class.

- A binary operator can be overloaded as a class member function with one argument or as a non-member function with two arguments (one of those arguments must be either an object of the class or a reference to an object of the class).
Restrictions on Operator Overloading

- Most of the C++ operators can be overloaded.
- Following operators are non-overloadable:
  \[ . \quad * \quad :: \quad ? : \]
- The precedence of operators cannot be changed by overloading, unless parenthesis is used to force the order.
- The associativity of an operator or the number of operands cannot be changed by overloading.
- It is not possible to create new operators.
- The meaning of how an operator works on built-in objects cannot be changed.
Overloading Stream-Insertion & Stream-Extraction

- The stream-insertion and stream-extraction operators can also be overloaded to perform input and output for user-defined types.

- When overloading `<<` and `>>`, the operator function needs to be as a friend (i.e. a non-member function).

- The overloaded `<<` must have a left operand of type `ostream&` (such as `cout`) in the expression `cout << classObject`.

- The overloaded `>>` must have a left operand of type `istream&` (such as `cin`) in the expression `cin >> classObject`.
Overloading `++` and `--`

- Let `d` be of type `Date`. The pre-increment expression:
  ```
  ++d;
  ```
- generates the member function call:
  ```
  d.operator++();
  ```
- whose prototype is:
  ```
  Date& Date::operator++();
  ```
Overloading ++ and --

- Let d be of type Date. The post-increment expression:

```
    d++;
```

- generates the member function call:

```
    d.opeartor++(0);
```

- whose prototype is:

```
    Date& Date::operator++(int);
```

- Note that the 0 is strictly a “dummy value” to make the argument list of operator++, used for post-incrementing.
Example of Overloading ++

Complex & Complex::operator++( )  // preincrement
{
    real += 1;
    return *this;                    // enables cascading
} // end operator++ function

Complex Complex::operator++(int)  // postincrement
{
    Complex temp = *this;
    real += 1;
    return temp;                    // enables cascading
} // end operator++ function
Example of Overloading --

Complex & Complex::operator--( ) // predecrement
{
    real -= 1;
    return *this;
} // end operator-- function

Complex Complex::operator--(int) // postdecrement
{
    int temp = *this;
    real -= 1;
    return temp;
}

} // end operator-- function
Example of Overloading =

Employee & Employee::operator=( const Employee &right )
{
    if ( right != this ) //check for self-assignment
    {
        strcpy(firstname, right.firstname);
        strcpy(lastname, right.lastname);
    }
    return *this;   // enables cascading
} // end operator= function
Converting Between Types

- It is often necessary to convert data of one type to data of another type.
- Certain conversions among built-in types are performed by the compiler. Programmers can force conversions by casting.
- The programmer may specify how to convert among user-defined types and built-in types.
- Such conversions can be performed with *conversion constructors* (or *conversion/cast operator*)—single argument constructors that turn objects of other types (including built-in types) into objects of a particular class.
- A conversion operator must be a non-static class member function (it can’t be a *friend* function).
Converting Between Types (cont.)

- The function prototype:
  
  ```
  A::operator char *() const;
  ```

  declares an overloaded cast operator function for creating a temporary `char *` object out of an object of user-defined type `A`.

- An overloaded cast operator does not specify a return type - the return type is the type to which the object is being converted.

- If `s` is a class object, when the compiler sees the expression `(char *)s`, it generates the call

  ```
  s.operator char *();
  ```

- Cast operator functions can be defined to convert user-defined types into built-in or other user-defined types.
Converting Between Types (cont.)

- The prototypes:
  
  ```cpp
  A::operator int() const;
  A::operator otherClass() const;
  ```

- Declare functions for converting an object of a user-defined type `A` into an integer and an object of a user-defined type `A` into an object of `otherClass`.

- When necessary, the compiler can call these functions to create temporary objects. For example, if an object `s` of a user-defined `String` class appears in a program at a location where an ordinary `(char *)` is expected, such as

  ```cpp
  cout << s;
  ```

- The compiler calls the cast operator function. With this cast operator, the stream-insertion should not be overloaded.
Type Conversion

- Implicit conversions can be controlled by means of three member functions:
  - **Single-argument constructors:** allow implicit conversion from a particular type to initialize an object.
  - **Assignment operator:** allow implicit conversion from a particular type on assignments.
  - **Type-cast operator:** allow implicit conversion to a particular type.

- To prohibit implicit type conversion use the keyword `explicit` in front of the constructor.
Copy Constructor

- What is a copy constructor?
  - A copy constructor is a member function which initializes an object using another object of the same class.
  - A copy constructor has the following general function prototype:
    
    ```cpp
    ClassName (const ClassName &old_obj);
    ```
Copy Constructor (cont.)

When is copy constructor called?

In C++, a Copy Constructor may be called in following cases:

1. When an object of the class is returned by value.

2. When an object of the class is passed (to a function) by value as an argument.

3. When an object is constructed based on another object of the same class.

4. When compiler generates a temporary object.
Copy Constructor (cont.)

- **When is user defined copy constructor needed?**
  - If we don’t define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member wise copy between objects.
  - We need to define our own copy constructor only if an object has pointers or any run time allocation of resource like file handle, a network connection..etc.
Copy Constructor (cont.)

- Default constructor does only shallow copy.
Copy Constructor (cont.)

- **Deep copy is possible only with user defined copy constructor.**
  - In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.
Copy constructor vs Assignment Operator

Which of the following two statements call copy constructor and which one calls assignment operator?

MyClass t1, t2;  // -----> (1)
MyClass t3 = t1;  // -----> (1)
t2 = t1;  // -----> (2)

(1) Copy constructor is called
(2) Assignment