Operator Overloading

- **Operator overloading** is a very important capability.
  - It allows you to make standard C++ operators, such as +, -, * and so on, work with objects of your own data types.
  - We want to write
    - if (box1 > box2)
  - instead of
    - if (IsGreaterThan(box1, box2))

- Let us recall some background of **function overloading** (Chapter 6).

Function Overloading

- Function overloading allows you to use the same function name for defining several functions as long as they each have different parameter lists.
- When the function is called, the compiler chooses the correct version according to the list of arguments you supply.
- The following functions share a common name, but have a different parameter list:
  - `int max(int array[], int len);`
  - `long max(long array[], int len);`
  - `double max(double array[], int len);`

Ex6_07.cpp on P.293

- Three overloaded functions of max()
- In main(), C compiler inspect the argument list to choose different version of functions.

Signature

- The signature of a function is determined by its name and its parameter list.
- All functions in a program must have unique signatures
- The following example is not valid overloading
  - `double max(long array[], int len);`
  - `long max(long array[], int len);`
- A different return type does not distinguish a function, if the signatures are the same.

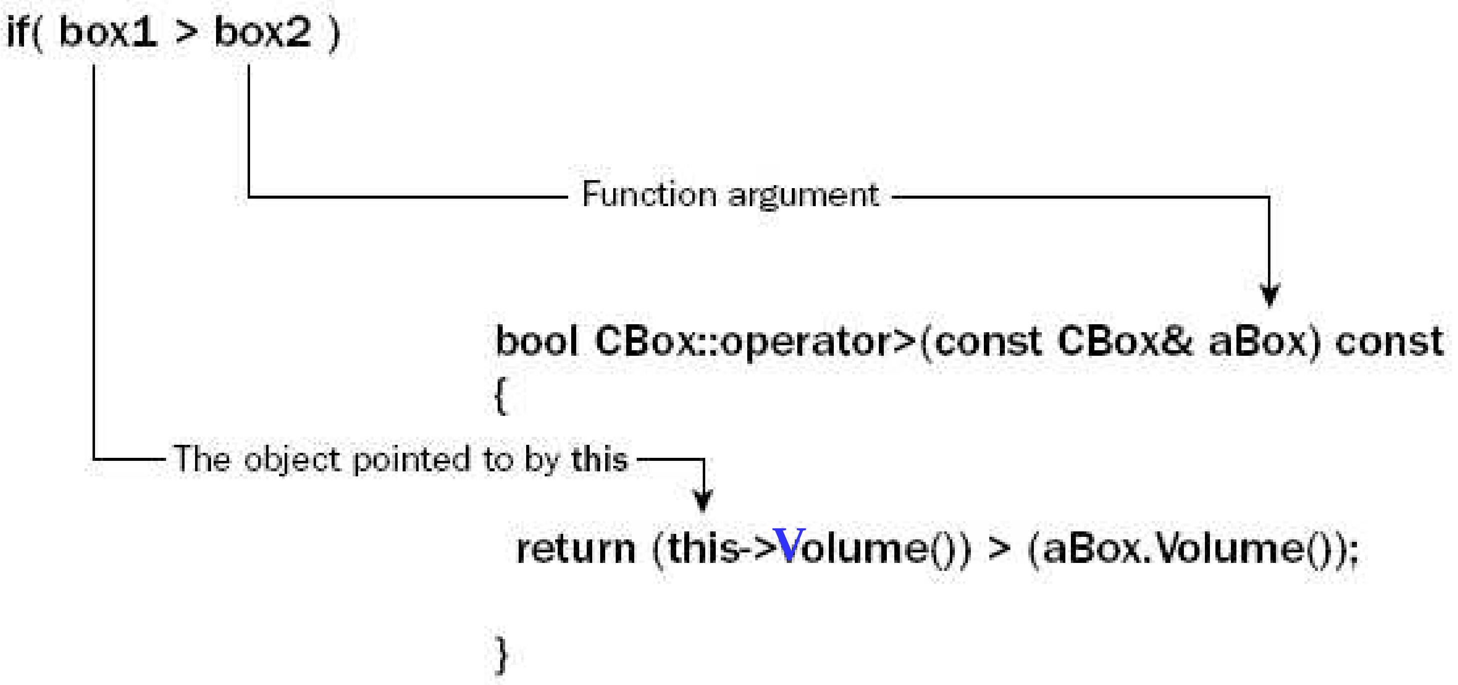
Implementing an Overloaded Operator

```
class CBox
{
  public:
    bool operator> (CBox& aBox) const;
}
```

- The word `operator` here is a keyword.
- You declare the `operator>()` function as const because it doesn’t modify any data members of the class. (P.369)

Using an Overloaded Operator

- if (box1 > box2)
  cout << “box1 is greater than box2”;
- if (box1.operator>(box2))



```
if( box1 > box2 )

bool CBox::operator>(const CBox& aBox) const
{
 return (this->Volume()) > (aBox.Volume());
}
```

Figure 8-3

Ex8_03.cpp on P.422

```
bool CBox::operator> (CBox& aBox) const
   {
      return this->Volume() > aBox.Volume();
   }
```

- The left operand is defined implicitly by the pointer **this**.
- The basic > operator returns a value of type int
  - 1 for true
  - 0 for false.
- It will be automatically converted to bool.

Overloading the Assignment Operator

- What's wrong with the default assignment?
  - It simply provides a member-by-member copying process, similar to that of the default copy constructor.
  - They suffer from the same problem, when some data members are allocated dynamically.

Fixing the Problem

```
CMessage& operator= (const CMessage& aMess)
{
  // Release memory for 1st operand
  delete [] pmessage;
  pmessage = new char [ strlen(aMess.pmessage) + 1];

  // Copy 2nd operand string to 1st
  strcpy(this->pmessage, aMess.pmessage);

  // Return a reference to 1st operand
  return *this;
}
```

Why Do You Need to Return Something?

- Consider this statement
  - motto1 = motto2 = motto3;
- The assignment operator is right-associative, so it translates into
  - motto1 = (motto2.operator=(motto3));
  - motto1.operator=(motto2.operator=(motto3));
- You must at least return a CMessage object.

Why Do You Need to Return a Reference?

- Consider another example
  - (motto1 = motto2) = motto3;
- This translates into
  - (motto1.operator=(motto2))   = motto3;
- If the return type is merely CMessage instead of a reference, a temporary copy of the original object is returned.
  - Then you are assigning a value to a temporary object!
  - Make sure that your return type is CMessage&.

Check Addresses, If Equal

- The first thing that the operator function does is to delete the memory allocated to the first object, and reallocate sufficient memory to accommodate the new string.
- What happens to this statement?
  - `motto1 = motto1`
- Add this checking:

```
if (this == &aMess)
      return *this;
```

Overloading the Addition Operator

- Suppose we define the sum of two CBox object as a CBox object which is large enough to contain the other two boxes stacked on top of each other.
- See Figure 8-4.

```
CBox CBox::operator+(const CBox& aBox) const
{
return CBox(
m_Length > aBox.m_Length ? m_Length : aBox.m_Length,
m_Width > aBox.m_Width ? m_Width : aBox.m_Width,
M_Height + aBox.m_Height);
}
```

- Ex8_06.cpp on P.434

Using Classes

- We want to pack *candy* into *candy boxes*, and pack *candy boxes* to *cartons*.
- The objects candy, candybox, carton, all belong to the `CBox` class.
- We are packing `CBox` objects into other `CBox` objects.

Basic Operations of the CBox Class

- Calculate the volume of a CBox
  - Volume()
- Compare the volumes of two CBox objects to determine which is the larger.
  - operator>()
- Compare the volume of a CBox object with a specified value
  - We have this for the > operator (P.414)
- Add two CBox object to produce a CBox object
  - operator+()
- Multiply a CBox object by an integer to provide a CBox object
- Determine how many CBox objects of a given size can be packed in another CBox object of a given size.
  - This is effectively division, so you could implement this by overloading the / operator.
- Determine the volume of space remaining in a CBox object after packing it with the maximum number of CBox objects of a given size.
  - Wasted spaced.

The Multiply Operation

- If n is even, stack the boxes side-by-side by doubling the m_Width value and only multiplying the m_Height value by half of n.

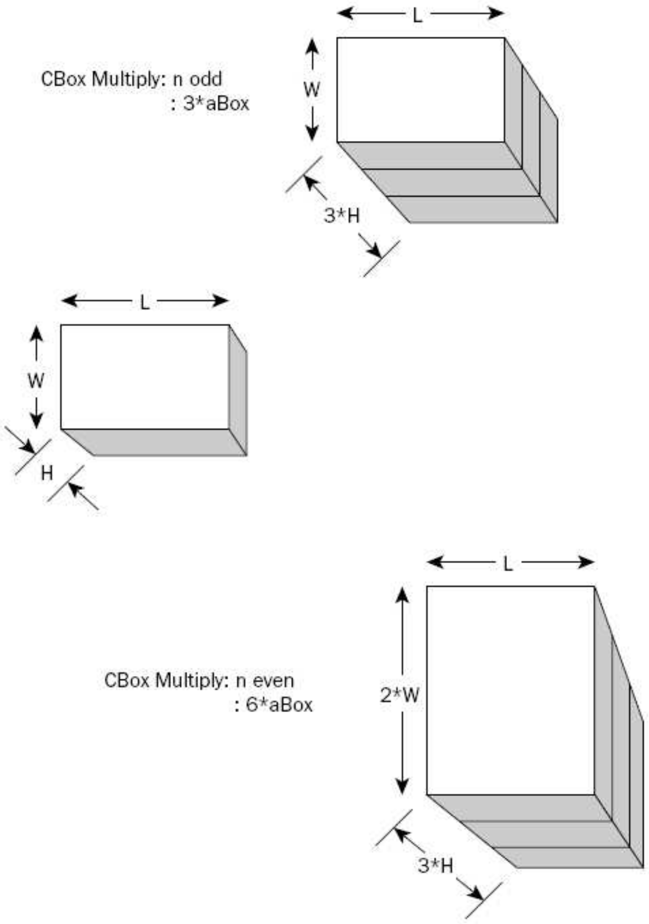


Figure 8-6

```
// CBox multiply operator this*n
CBox operator*(int n) const
{
  if (n % 2)
     return CBox(m_Length, M_Width,
  n*m_Height); // n odd
  else
     return CBox(m_Length, 2.0*m_Width,
  (n/2)*m_Height); // n even
}
```

The Division Operation

- Correct some mistakes in Figure 8-7.

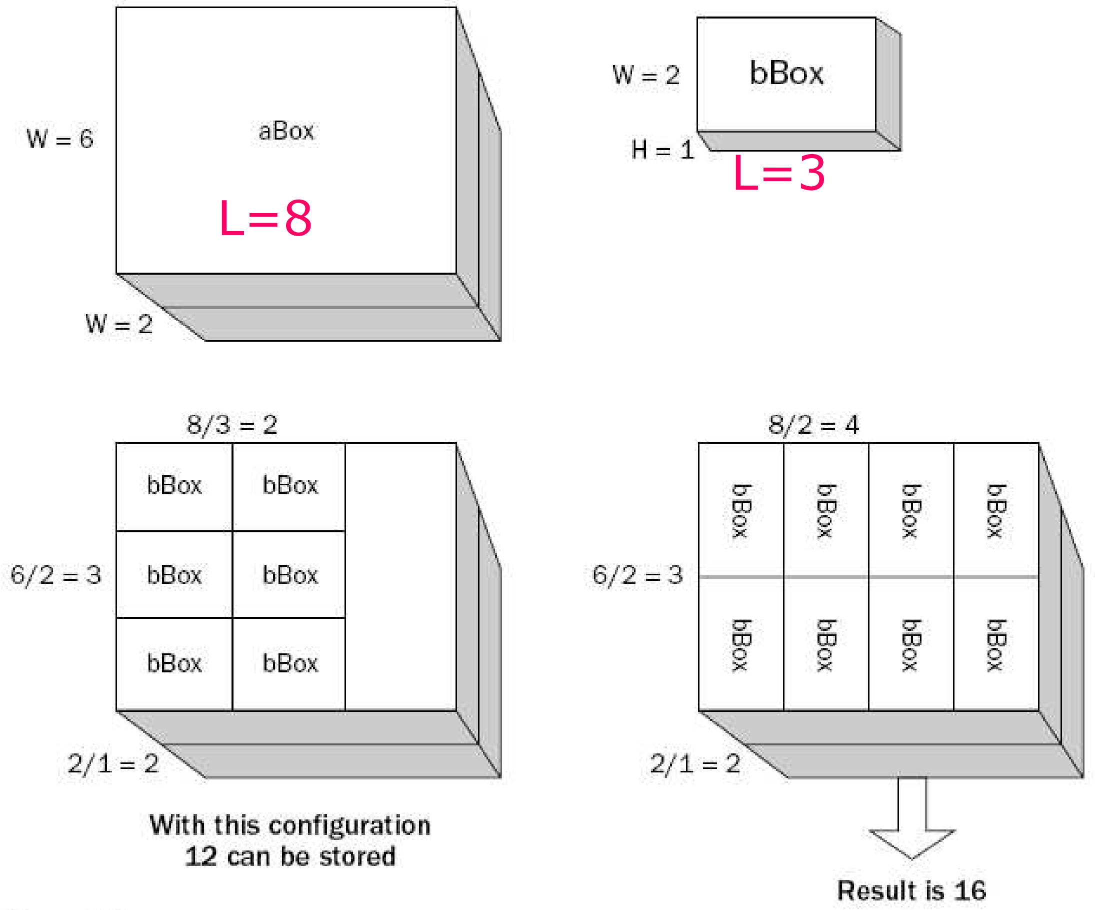


Figure 8-7

Member Function `operator/()`

- P.451

```
int operator/(const CBox& aBox)
{
   int tc1 = 0;
   int tc2 = 0;

   tc1 = static_cast<int>((m_Length / aBox.m_Length)) *
        static_cast<int>((m_Width / aBox.m_Width));
   tc2 = static_cast<int>((m_Length / aBox.mWidth)) *
        static_cast<int>((m_Width / aBox.m_Length));

   return
        static_cast<int>((m_Height/aBox.m_Height)*(tc1>tc2 ?
                tc1 : tc2));
}
```

Member Function `operator%()`

- It would be easy to check the remaining space using the functions you have already defined:

```
// Operator to return the free volume in a packed CBox
double operator%( const CBox& aBox, const CBox& bBox)
{
  return aBox.Volume() - (aBox / bBox) * bBox.Volume();
}
```