

Operator Overloading

- Which C++ operators can be overloaded?
- Implement functions in your classes that overloaded operators
- Overloaded operator functions as member function in class
- Assignment operators
- Define type conversions as operator functions
- Smart pointers

Principals of Operator Overloading

- Reasonability
- Consistency
- Instinct
- Cannot create a new operator
- Preserve precedents
- Preserve unary, binary, trinary

Operators That Can and Cannot Be Overloaded

- Every operator list in Appendix D, excluding
 - Domain operator – ::
 - Conditional operator – ?:
 - Direct access operator – .
 - Dereference operator – .*
 - sizeof operator – sizeof
 - Preprocessor symbol – # or ##

The operator<()

```
class Box {  
    public:  
        bool operator<(const Box& aBox) const;  
  
        // The rest of the Box class  
};
```

- Program 14.1

The diagram illustrates the mapping of code in Program 14.1 to the operator function definition. It shows the following code snippets:

```
if( box1 < box2 )  
  
bool Box::operator<(const Box& aBox) const {  
    return this->volume() < aBox.volume();  
}
```

Arrows indicate the following mappings:

- An arrow from `box1` in the `if` statement to `this` in the `operator<` function, labeled "The object pointed to by this".
- An arrow from `box2` in the `if` statement to `aBox` in the `operator<` function, labeled "The argument to the operator function".

Implementing Full Support for an Operator<()

- Previous define of operator<() is able to implement box1 < box2, but not
- box1<25.6 or 10<box2

```
inline bool operator<(const double d) const  
{ return volume() < d; }  
inline bool operator<(const double d, const Box& b)  
{ return d < b.volume(); }
```

- Program 14.2

Binary OperatorX()

- Member function
 - Type operatorX(Type RightOperand);
 - Type operatorX(Other_Type RightOperand);
- Non-member function
 - Type operator X(Other_Type leftOperand, Type RightOperand);

Compiler Supported Member Functions

- Default constructor
- Destructor
- Copy constructor
- Assignment operator

Copy Constructor and Assignment Operator

- New space is generated during copy constructor, no new space is generated during assignment
- Copy constructor is called when
 - `Box aBox(bBox); // Box aBox = bBox;`
 - `TruckLoad aLoad1(aLoad2);`
- Assignment operator is called when
 - `aBox = bBox;`
 - `aLoad1 = aLoad2;`

Overloading the Assignment Operator

```
Box box1;  
Box box2(10, 10, 10);  
box1 = box2;           // Call the assignment operator
```

```
class Data {  
    public:  
        int value;  
};
```

Compiler
supports



```
class Data {  
    public:  
        int value;  
  
        Data(){}  
        ~Data(){}  
  
        Data(const Data& aData) : Value(aData.value) {}  
  
        Data& operator=(const Data& aData) {  
            value = aData.value;  
            return *this;  
        }  
};
```

Assignment Operators

```
load1 = load2 = load3;
```

```
load1 = (load2 = load3);
```

```
load1.operator=(load2.operator=(load3));
```

Overloading the Assignment Operator

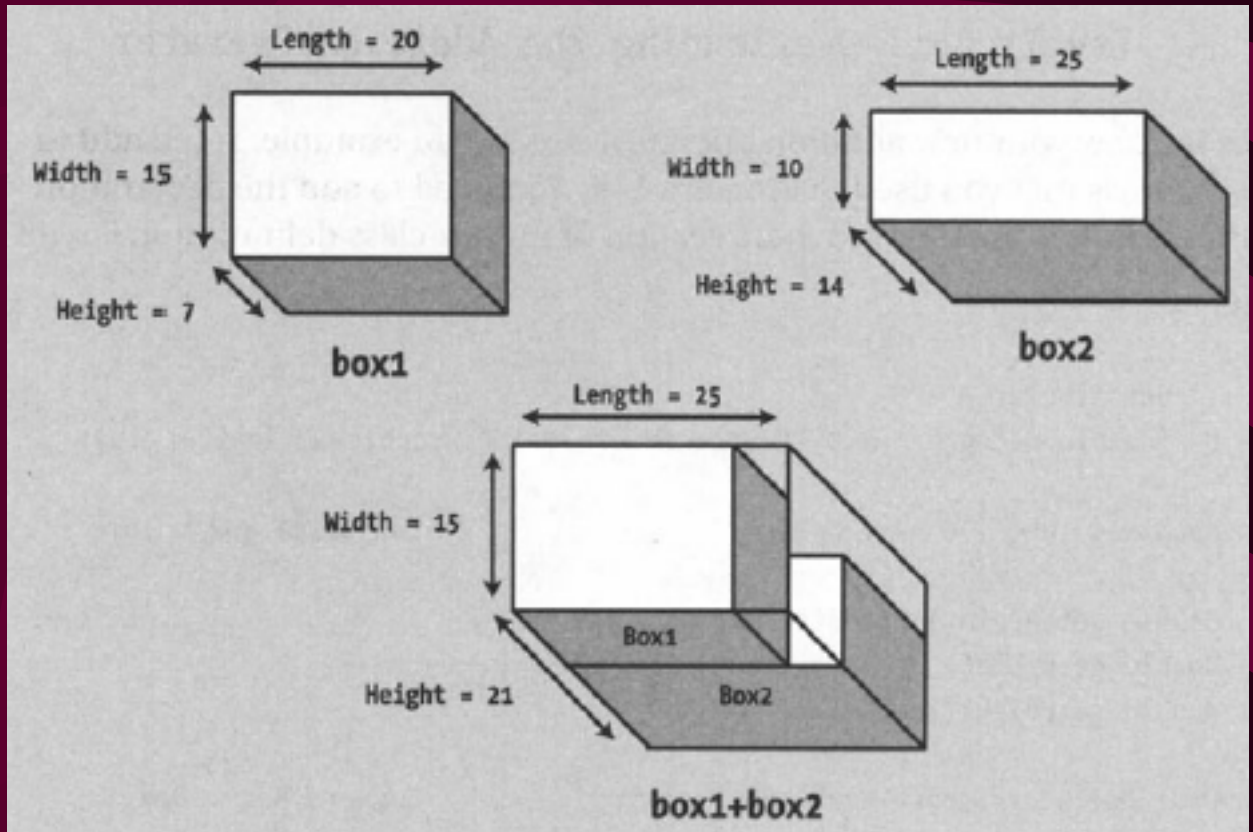
```
TrunckLoad& TrunckLoad::operator=(const TrunckLoad& load)
{
    if (this == &load) return *this;
    while (pCurrent = pHead)
    {
        pHead = pHead->pNext;
        delete pCurrent;
    }
    pHead = pTail = pCurrent = 0;
    if (load.pHead == 0) return;
    package* pTemp = load.pHead;
    do { addBox(pTemp->pBox); } while (pTemp = pTemp->pNext);
    return *this;
}
```

Process of Overloading the Assignment Operator

- Remove the memory using in the left operand, create identical memory space of the right operand for the left operand
- Copy the content of the right operand to the left operand
- Return *this
- Program 14.3

Overloading the Arithmetic Operators

- $+$, $-$, $*$, $/$, $\%$
- Program



Implementing One Operator in Terms of Another

- Such as operator+ can be applied in operator+=

- Program 14.5a

```
// Function to add two Box objects
inline Box Box::operator+(const Box& aBox) const {
    return Box(*this) += aBox;
}
```

```
// Overloaded += operator
inline Box& Box::operator+=(const Box& right) {
    length = length > right.length ? length : right.length;
    width = width > right.width ? width : right.width;
    height += right.height;
    return *this;
}
```

```
inline bool Box::operator>=(const Box& aBox) const {
    return !(*this<(aBox));
}
```

Rules

- return by value
 - ex: +, -, *, /, %
 - a = b+c;
 - return temp object
- return by reference
 - ex: +=, -=, /=, []
 - a+=b;
 - return *this

Overloading The Subscript Operator []

```
class TruckLoad
{ public:
  // For left value
  Box& operator[](int index);
  // For right value
  const Box& operator[](int index) const;
  ...
private:
  class Package { ...};
};
```


Revised The Package Class

```
class Package
{ public:
    Box theBox;
    Package* pNext;
    void setNext(Package* pPackage);
    Package(const Box& NewBox);
    ~Package();
};
```

Subscript Operator []

```
Box& TruckLoad::operator[](int index)
{
    if (index<0)
    { cout << endl << "Negative index"; exit(1); }
    Package* pPackage = pHead;
    int count = 0;
    do
    { if (index == count++) return pPackage->theBox;
    } while (pPackage = pPackage->pNext);
    cout << endl << "Out of range index";
    exit(1);
} // Revised Program 14.5
```

Lvalues and Overloaded Subscript Operator []

- To deal with
 `load[0] = load[1];`
- Solution
 - Redefine [] operator to return a reference Box
 - `Box& TruckLoad::operator[](int index);`
 - `const Box& TruckLoad::operator[](int index) const;`

Overloading Type Conversions

```
class Object
{
public:
    operator Type();
    // conversion from Object to Type
};
```

Type Conversion From TruckLoad to Box

```
class TruckLoad
{
public:
    operator Box() const;
    // reset of TruckLoad class definition
};
```

Type Conversion from Box to double

```
class Box
```

```
{
```

```
public:
```

```
    operator double() const
```

```
    { return volume(); }
```

```
    // reset Box class definition
```

```
};
```

```
Box theBox;
```

```
double boxVolume = theBox; // double(theBox)
```

```
double total = 10 + static_cast<double>(theBox);
```

Ambiguity with Conversions

- Make constructor as a type conversion, should be carefully use

```
class Type2
{
    Type2(const Type1& theObject);
    // constructor converting Type1 to Type2
    operator Type1();
    // conversion from Type2 to Type1
};
```

- Use 'explicit' in constructor

Overloading ++, -- Operators

- Prefix ++ and Postfix ++

```
class Object
{
public:
    Object& operator++();    // prefix
    const Object operator++(int); // postfix
};
```


Smart Pointer

- We are able to overload the dereference operator, `*`, and the indirect operator, `->`
- Define a type that represents a smart pointer, something that behaves like a pointer but is really a class object
- Smart pointers referred to as **class iterators**

BoxPtr Class

```
class BoxPtr
{ public:
    BoxPtr(TruckLoad& load); // constructor
    Box& operator*() const;   // overload *
    Box* operator->() const;  // overload ->
    Box* operator++();       // prefix increment
    const Box* operator++(int); // postfix increment
    operator bool();         // conversion to bool
private:
    Box* pBox; TruckLoad& rLoad; ...
}; // Revised Program 14.6
```

Overloading Operators new and delete

- Memory allocation and deallocation faster and more economical for a particular class of objects

```
class Data
{public:
    void* operator new(size_t size);
    void operator delete(void* obj, size_t size);
};

void* operator new(size_t size)
{ pSpace = ::new char(size);
} // allocated by global new
```

Fine-Tuning for Overloading Operators

| overloading operators | Return | Argument passed by | Function |
|-----------------------|----------------------|--------------------|-----------|
| Arithmetic +,-,*,/,% | by value | const reference | const |
| Assignment =,+=,-= | by reference (*this) | const reference | non-const |
| Comparison <,== | by value (boolean) | const reference | const |
| Unary prefix ++,-- | by reference (*this) | none | non-const |
| Unary -,+ | by value | none | const |
| Subscript [] | by reference | integer | non-const |
| << | by reference | const reference | non-const |
| >> | by reference | const reference | const |

Refinement of prefix ++

```
airtime& operator++ ()  
{  
    ++minutes;  
    if(minutes >= 60)  
    {  
        ++hours;  
        minutes -= 60;  
    }  
    return *this;  
}
```

Refinement of postfix ++

```
airtime operator++ (int)
{
    airtime temp(hours, minutes);
    ++minutes;
    if(minutes >= 60)
    {
        ++hours;
        minutes -= 60;
    }
    return temp;
}
```