CSE 165/ENGR 140
Intro to Object Orient Program
Lecture 19 – Operators
Announcement

- Lab #7
  - Due 4/9 at 11:59PM

- Quiz: 4/16

- Final project out this Friday (in UCMCROPS)
  - Due date: 5/11 (Monday) at 11:59PM
  - Presentation date: 5/12 (Tuesday) at 3:00PM

- Reading assignment:
  - Ch. 16
Most operators can be overloaded
  ◦ they provide specific functionality for your objects when used with operators

Operators used in expressions with built-in data types cannot be changed
  ◦ only expressions with user-defined types can
Operators

- Very Convenient
  - but they just help with syntax, no new functionality is given

- List of operators that can be overloaded:

```
+  -  *  /  =  <  >  +=  -=  *=  /=
<<  >>  <<=  >>=  ==  !=  <=  >=  ++  --  %
&  ^  !  |  ~  &=  ^=  |=  &&  ||  %=
[ ]  ()  ,  ->*  ->  new  delete  new[]  delete[]
```

- Do not forget:
  - operators are always overloaded, therefore, the types of the objects affected by an operator will tell which operator version will be used.
Operators

- Types of Operators
  - Unary (one argument)
  - Binary (two arguments)

- Ways of overloading
  - as a global function
    - Unary operators will have one argument
    - Binary operators will have two arguments
  - as a member function
    - Unary operators will have no arguments
    - Binary operators will have one argument
Example: overloading + and +=

```cpp
#include <iostream>
using namespace std;

class Integer {
    int i;

public:
    Integer(int ii) : i(ii) {} 

    const Integer operator+(const Integer& rv) const {
        cout << "operator+" << endl;
        return Integer(i + rv.i);
    }

    Integer& operator+=(const Integer& rv) {
        cout << "operator+=" << endl;
        i += rv.i;
        return *this;
    }
};

int main() {
    Integer ii(1), jj(2), kk(3);
    kk += ii + jj;
}
```

Note: you are responsible for returning the needed type for the operator to work as you want: in the example, + uses a const object, += needs a reference.
Unary Operators as Global functions

// Non-member functions:
class Integer {
    long i;
    Integer* This() { return this; }

public:
    Integer(long ll = 0) : i(ll) {}
    friend const Integer& operator+ (const Integer& a);
    friend const Integer operator~ (const Integer& a);
    friend Integer* operator&(Integer& a);
    friend int operator! (const Integer& a);
    friend const Integer& operator++ (Integer& a);
    friend const Integer operator++ (Integer& a, int);
};

const Integer& operator+(const Integer& a) { return a; } // Unary + has no effect
const Integer operator~(const Integer& a) { return Integer(~a.i); }
Integer* operator&(Integer& a) { return a.This(); } // &a is recursive!
int operator!(const Integer& a) { return !a.i; }
const Integer& operator++(Integer& a) { a.i++; return a; }
const Integer operator++(Integer& a, int) { Integer before(a.i); a.i++; return before; }

// Test some of the overloaded operators:
void f(Integer a) {
    +a;
    ~a;
    Integer* ip = &a;
    ++a;
    a++;}

// Operators overloaded as class members (they are easier to write in most cases):
class Byte {
    char b;
public:
    Byte(unsigned char bb = 0) : b(bb) {}

    const Byte& operator+() const { return *this; }
    const Byte operator-() const { return Byte(-b); }
    Byte operator!() const { return Byte(!b); }
    const Byte& operator++() { b++; return *this; }// Prefix
    const Byte operator++(int) { Byte before(b); b++; return before; }// Postfix
};

void g ( Byte b ) {
    +b;
    -b;
    ++b;
    b++;}
class Integer {
    long i;
public:
    Integer(long ll = 0) : i(ll) {}  
    friend const Integer operator+(const Integer& left, const Integer& right);
    friend const Integer operator*(const Integer& left, const Integer& right);
    friend const Integer operator<<(const Integer& left, const Integer& right);
    friend Integer& operator+=(Integer& left, const Integer& right);
    friend bool operator==(const Integer& left, const Integer& right);
    friend int operator||(const Integer& left, const Integer& right);
};

const Integer operator+(const Integer& left, const Integer& right) {  
    return Integer(left.i + right.i);
}
const Integer operator*(const Integer& left, const Integer& right) {  
    return Integer(left.i * right.i);
}
    // etc
// Member functions (implicit "this"):

```cpp
class Byte {
    unsigned char b;

public:
    Byte(unsigned char bb = 0) : b(bb) {}
    // No side effects: const member function:
    const Byte operator+ (const Byte& right) const {
        return Byte(b + right.b);
    }

    const Byte operator* (const Byte& right) const {
        return Byte(b * right.b);
    }

    Byte& operator+= (const Byte& right) {
        if(this == &right) {/* self-assignment */}
        b += right.b;
        return *this;
    }

    Byte& operator/= (const Byte& right) {
        require(right.b != 0, "divide by zero");
        if(this == &right) {/* self-assignment */}
        b /= right.b;
        return *this;
    }
}
```

(continued in the next slide)
Binary Operators as Members

Byte& operator%(const Byte& right) {  
    require(right.b != 0, "modulo by zero");  
    if(this == &right) {/* self-assignment */}  
    b %= right.b;  
    return *this;  
}

// Conditional operators return true/false (or int):
int operator==(const Byte& right) const {  
    return b == right.b;  
}
int operator!=(const Byte& right) const {  
    return b != right.b;  
}
int operator>(const Byte& right) const {  
    return b > right.b;  
}
int operator<=(const Byte& right) const {  
    return b <= right.b;  
}
int operator>=(const Byte& right) const {  
    return b >= right.b;  
}
int operator&&=(const Byte& right) const {  
    return b && right.b;  
}
int operator|=(const Byte& right) const {  
    return b || right.b;  
};
Special and Unusual Operators

- Copy Operator
  - Can only be defined as a member
  - Expected functionality is to make a copy of the object

- Operators [], and ()
  - Often used as an accessor operator (to simulate the functionality of array indexing)

- Operator “,” can also be overloaded
  - you can have expressions like: a = b,c;
There is also operator \( \Rightarrow \)\( ^* \)
- see examples in the book, but we will not be using this operator for anything

Operator \( \Rightarrow \) can also be overloaded
- expected to simulate the behavior of pointers
- typical example: to implement “smart pointers”
Smart Pointer Example 1/2

// Generic Obj class for our example:
class Obj {
    static int i, j;
public:
    void f() const { cout << i++ << endl; }
    void g() const { cout << j++ << endl; }
};

// Static member definitions:
int Obj::i = 47;
int Obj::j = 11;

// Container of Obj*:
class ObjContainer {
    vector<Obj*> a;
public:
    void add (Obj* obj) { a.push_back(obj); }
    friend class SmartPointer;
};
class SmartPointer {
    ObjContainer& oc;
    unsigned index;
public:
    SmartPointer(ObjContainer& objc) : oc(objc) { index = 0; }

    bool operator++() { // Prefix (return value indicates end of list)
        if(index >= oc.a.size()) return false; // test bounds (the pointer is smart!)
        if(oc.a[++index] == 0) return false;
        return true;
    }
    bool operator++(int) { // Postfix
        return operator++(); // Use prefix version
    }
    Obj* operator->() const {
        ExitWithError (oc.a[index] != 0, "Zero value returned by SmartPointer::operator->()" );
        return oc.a[index]; // the index is valid (the pointer is smart!)
    }
};

int main() {
    const int sz = 10;
    Obj o[sz];
    ObjContainer oc;
    for (int i = 0; i < sz; i++ ) oc.add(&o[i]); // Fill it up
   SmartPointer sp(oc); // Create an “iterator”
    do { sp->f(); // Pointer dereference operator call
        sp->g();
    } while(sp++); // Pointer increment operator used
}
Special and Unusual Operators

- What cannot be overloaded
  - operator . (dot) cannot be overloaded
  - there is no exponential operator
  - no user-defined operators
  - no change of precedence rules possible

- Extra
  - use `const` as much as possible
  - returning a value in most cases is optional
    - Ex: copy operator can return a value in order to make `a=b=c;` work.
class IntArray {
    enum { sz = 5 };  
    int i[sz];
public:
    IntArray() { memset(i, 0, sz* sizeof(*i)); }  

    int& operator[](int x) {
        require(x >= 0 && x < sz, "IntArray::operator[] out of range");
        return i[x];
    }

    friend ostream& operator<< (ostream& os, const IntArray& ia) {
        for(int j = 0; j < ia.sz; j++) {
            os << ia.i[j];
            if(j != ia.sz -1) os << ", ";
        }
        os << endl;
        return os;
    }

    friend istream& operator>> (istream& is, IntArray& ia) {
        for(int j = 0; j < ia.sz; j++) is >> ia.i[j];
        return is;
    }

};

int main() {
    stringstream input("47 34 56 92 103");
    IntArray I;
    input >> I;
    I[4] = -1; // Use overloaded operator[]
    cout << I;
}
Copy/Assignment Operator

- Be careful with the Copy Operator!
  - If you do not give one the compiler will do a “bit by bit” assignment of the members.

- What to do if your class manages a pointer `pt` to some allocated memory and a copy `a=b` happens?
  - **deep copy**: create an independent copy: allocate a new memory space for `a` and copy the contents of `b.pt` to `a.pt`
  - **shallow copy**: rely on reference counting, for example use our Shared class and do: `a.pt=b.pt; a.pt->ref()`
You can make operators to convert one type to another!

- They are automatically called when the conversion is clear.
- You can use “type cast notation” to clarify the type conversion.

```cpp
class NameId
{
public:
    // some other members needed here...
    /* type conversion operator: returns the associated string or "" if the name is undefined. */

    operator const char* ( int id )
    {
        return id==_undefined? "": internal_table[id];
    }
};
```
Final Notes

- The book has many more examples on details about overloading operators
  - We only cover the basic functionality of overloading operators.
  - Focus on learning the basic operations so that you can do Lab #10.
  - But, of course, read the book to learn about all the other things that are possible!

- About the shallow and deep copy example
  - It is an important choice in respect to memory management design (not really operators).