Operator Overloading in C++
++, −, (), ->

October 21, 2004
Outline

- Background
- Increment and decrement: `++`, `–`
- Function call operator: `()`
- Dereferencing (arrow operator): `->`
Background

"When I use a word it means just what I choose it to mean - neither more nor less." – Humpty Dumpty
Chapter 11

Issues:
- Binary and unary operators
- Predefined meanings for operators
- User-defined meanings for operators
- Operators and namespaces
- Member and nonmember operators
- Conversion operators
- Ambiguity resolution
- Friends
- Members and friends
- Large objects
Chapter 11

Issues (cont):
- Assignment and initialization
- Subscribing ([[]])
- Function call
- Dereferencing
- Increment and decrement

Examples:
- complex
- String
## What can we overload?

We can overload:

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>%</th>
<th>^</th>
<th>&amp;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>~</td>
<td>!</td>
<td>=</td>
<td>&lt;</td>
<td>&gt;</td>
<td>+=</td>
</tr>
<tr>
<td>-=</td>
<td>*=</td>
<td>/=</td>
<td>%=</td>
<td>^=</td>
<td>&amp;=</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td>&lt;&lt;= =</td>
<td>!=</td>
<td>&lt;=</td>
<td>&gt;=</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>&amp; &amp;</td>
<td></td>
<td></td>
<td>++</td>
<td>--</td>
<td>-&gt;*</td>
</tr>
<tr>
<td>-&gt;</td>
<td>[]</td>
<td>()</td>
<td>new</td>
<td>new[]</td>
<td>delete</td>
<td>delete[]</td>
</tr>
</tbody>
</table>

But we cannot:

:: . . . . *
What we can and cannot?

We cannot:

- define new operator tokens (Ex: **) 
- change the arity (Ex: make ! binary) 
- change the priority 

But we can:

- not maintain the usual equivalence between some operators 

Ex:

```c++
Y* p;
p->m == (*p).m == p[0].m
```
We can overload both pre- and post- version of increment and decrement:

class Foo {
    Foo &operator++(); // pre-increment
    Foo operator++(int); // post-increment
}

Originally C++ only allowed the prefix version to be overloaded.

The int argument from post- version is fake.
Function call overloading

- is an exception among operators: it can have any numbers of arguments (even none)
- it allows objects to look and act like functions (functors)

Ex:

```cpp
class Foo {
    int operator()(); // no arguments
    int operator()(int x) // one double argument
};
```
Dereferencing

- is used to access an element of a class (data or method) through a pointer (Ex: o->x, o->f())
- is an operator with only one argument! (the one from the left) Ex:

```cpp
class Foo1 {
    int x;
}

class Foo2 {
    Foo1 *operator->();
}

Foo2 f2;
f2->x
```
Advice

From *The C++ Programming Language, Third Edition* by Bjarne Stroustrup:

- Define operators primarily to mimic conventional usage;
- For large operands, use const reference argument types;
- For large results, consider optimizing the return;
- Prefer the default copy operations if appropriate for a class;
- Redefine or prohibit copying if the default is not appropriate for a type;
- Prefer member functions over nonmembers for operations that need access to the representation;
Advice (cont)

From *The C++ Programming Language, Third Edition* by Bjarne Stroustrup:

💡 Prefer nonmember functions over members for operations that do not need access to the representation;

💡 Use namespaces to associate helper functions with “their” class;

💡 Use nonmember functions for symmetric operators;

💡 Use () for subscripting multidimensional arrays;