CIS 330
C++ and Unix

Lecture 15

Operator Overloading

Operator Overloading

Syntactic sugar - simply another way of calling a function

Instead of arguments appearing inside (…), they surround the operator

You can define new operators that work with specific classes

- For example, “adding” two classes may have some semantic meaning
- Define the “+” operator to do this (which will then call a function)

Warning

Do not overuse operator overloading

Only use it if it makes sense, AND makes it easier to read your code

You cannot overload operators that are used with built-in types

- For example, you cannot change the meaning of + in `5 + 10`

Definition

Define it like a regular function, but with

`operator@`

Where `@` is the operator you want to overload

Example

```
class Integer {
  int i;
public:
  Integer(int ii) : i(ii) {}
  const Integer operator+(const Integer& rv) const {
    return Integer(i + rv.i);
  }
  Integer& operator+=(const Integer& rv) {
    i += rv.i;
    return *this; // l-value
  }
};
int main() {
  cout << "built-in types:" << endl;
  int i = 1, j = 2, k = 3;
  k += i + j;
  cout << "user-defined types:" << endl;
  Integer ii(1), jj(2), kk(3);
  kk += ii + jj;
}
```

Example

```
class Integer {
  int i;
public:
  Integer(int ii) : i(ii) {}
  const Integer operator+(const Integer& rv) const {
    Integer(i + rv.i);
  }
  Integer& operator+=(const Integer& rv) {
    i += rv.i;
    return *this; // l-value
  }
};
int main() {
  cout << "built-in types:" << endl;
  int i = 1, j = 2, k = 3;
  k += i + j;
  cout << "user-defined types:" << endl;
  Integer ii(1), jj(2), kk(3);
  kk += ii + jj;
}
```

The `operator+` produces a new Integer (a temporary) that is used as the `rv` argument for the `operator+=`.

The temporary is destroyed when it is no longer needed

Return Value

Member function **operator** is called for the object on the left-hand side (LHS) of the operator

The argument will be the right-hand side (RHS) of the operator

For non-conditional operators (conditionals usually return a boolean), you will almost always want to return an object, or a reference to an object of the same class/type

- If they are NOT the same type, the interpretation of what it should produce is up to you (e.g., for classes that store `char` and `int`, what should `char + int` produce?)

Overloadable Operators

You can overload almost all the operators in C

- But their use is fairly restrictive. For example,
  - you cannot combine operators that have no meaning in C (e.g., ** to represent exponentiation),
  - you cannot change the order of evaluation precedence,
  - you cannot change the number of arguments required

Two methods

- Define it as a global functions (and use `friend` to allow access)
- Define it as a member function

Unary Operators (Global)

```
class Integer {
  long i;
  Integer* This() { return this;
}
public:
  Integer(long ll = 0) : i(ll) {}
  // No side effects takes
  // const& argument:
  friend const Integer&
    operator+(const Integer& a);
  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);
  // Side effects have non-const&
  // argument:
  // Prefix:
  friend const Integer&
    operator++(Integer& a);
  // Postfix:
  friend const Integer
    operator++(Integer& a, int);
  // Prefix:
  friend const Integer&
    operator--(Integer& a);
  // Postfix:
  friend const Integer
    operator--(Integer& a, int);
};
```

Unary Operators (Global)

```
const Integer& operator+(const Integer& a) {
  cout << "+Integer\n";
  return a; // Unary + has no effect
}
const Integer operator-(const Integer& a) {
  cout << "-Integer\n";
  return Integer(-a.i); // Create a new
Integer object
}
const Integer operator~(const Integer& a) {
  cout << "~Integer\n";
  return Integer(~a.i);
}
Integer* operator&(Integer& a) {
  cout << "&Integer\n";
  return a.This(); // what happens if we make
this const?
}
int operator!(const Integer& a) {
  cout << "!Integer\n";
  return !a.i;
}
```

```
const Integer& operator++(Integer& a) {
  cout << "++Integer\n";
  a.i++; // a is changed
  return a; // a is returned
}
const Integer operator++(Integer& a, int) {
  cout << "Integer++\n";
  Integer before(a.i);
  a.i++; // a is changed
  return before; // copy of a before change
is returned
}
const Integer& operator--(Integer& a) {
  cout << "--Integer\n";
  a.i--;
  return a;
}
const Integer operator--(Integer& a, int) {
  cout << "Integer--\n";
  Integer before(a.i);
  a.i--;
  return before;
}
```

Unary Operators (Member)

Why are these different?

No argument

```
class Byte {
  unsigned char b;
public:
  Byte(unsigned char bb = 0) : b(bb) {}
  const Byte& operator+() const {
    cout << "+Byte\n"; return *this;
  }
  const Byte operator-() const {
    cout << "-Byte\n"; return Byte(-b);
  }
  const Byte operator~() const {
    cout << "~Byte\n"; return Byte(~b);
  }
  Byte operator!() const {
    cout << "!Byte\n"; return Byte(!b);
  }
  Byte* operator&() {
    cout << "&Byte\n"; return this;
  }
  const Byte& operator++() { //pre
    cout << "++Byte\n";
    b++; return *this;
  }
  const Byte operator++(int) { //post
    cout << "Byte++\n";
    Byte before(b);
    b++; return before;
  }
  const Byte& operator--() { //pre
    cout << "--Byte\n";
    --b; return *this;
  }
  const Byte operator--(int) { //post
    cout << "Byte--\n";
    Byte before(b);
    --b; return before;
  }
};
```

++ and --

You want to be able to call different functions, depending on whether it's `++a` (pre) or `a++` (post)

- `++a` generate a call to `operator++(a)`
- `a++` generate a call to `operator++(a, int)`
- This is done simply to differentiate the functions - the second int for `a++` does not get used

Binary Operators (Global)

```
class Integer {
  long i;
public:
  Integer(long ll = 0) : i(ll) {}
  // Operators that create new,
  // modified value:
  friend const Integer
    operator+(const Integer& left,
              const Integer& right);
 friend const Integer
    operator<<(const Integer& left,
               const Integer& right);
…
```

```
 // Assignments modify & return
lvalue:
  friend Integer&
    operator+=(Integer& left,
               const Integer& right);
…
 // Conditional operators return
true/false:
  friend int
    operator==(const Integer& left,
               const Integer& right);
…
void print(std::ostream& os) const {
os << i; }
};
```

Binary Operators (Global)

For example, `(a+=1)++;` is legal, but `(++a)++;` is NOT legal in C++

```
// Operators that create new,
// modified value:
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);
}
```

```
// Assignments modify & return lvalue:
Integer& operator+=(Integer& left,
                    const Integer& right)
{
   if(&left == &right) {
   /* self-assignment */}
   left.i += right.i;
   return left;
}
// Conditional operators return
true/false:
int operator==(const Integer& left,
               const Integer& right) {
    return left.i == right.i;
}
```

Binary Operator (Member)

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

Binary Operator

`operator=` is ONLY allowed to be a member function

Assignments operators (e.g., `operator+=`) have code to check for self-assignment (although it does not do anything)

- This is a general guideline - there are cases where self-assignment is required (e.g., `A+=A` to add to itself)
- However, for `operator=` (depending on what the "=" means) you may have to handle self-assignment as a separate case

Summary

Unary

- Global vs. Member
  - `friend const Integer operator-(const Integer& a);` vs.
  - `const Byte operator-() const`
- Differentiate pre- and post- operator using different function definition
  - `friend const Integer& operator++(Integer& a);`
  - `friend const Integer operator++(Integer& a, int);`

Binary

- Global vs. Member
  - `friend Integer& operator+=(Integer& left, const Integer& right);` vs.
  - `Byte& operator+=(const Byte& right);`
- `operator=` is only allowed as a member overloaded function

Arguments and Return Values

You can pass it in any way you want

- You just deal with bugs later

However, the better practice is to restrict what you can do with them depending on what the operator requires

- For example, if you only need to **read** from the arguments, default to passing it as **const reference**
  - Ordinary operators like +, −, conditionals, typically do not change their arguments, so need to be passed in as const reference
  - If they are member functions make it a `const` member function

For assignment operators (e.g., +=, =) that change the left-hand argument, the left-hand side is NOT a `const`

Arguments and Return Values

Type of return value depends on the expected "meaning" of the operation

All assignment operators modify the left-hand value (l-value)

- To allow this to be used in **chained** expression (e.g., `a = b= c;`), it is expected that **reference to the l-value that was modified** is returned
- Since `a = b = c;` is read from left to right by the compiler, you CAN have it return `const`, but if you want to perform an operation on it (e.g., `(a = b).func();` to call `func()` on `a` after assigning `b` to it), the return value should be non-`const` reference (remember that you can't call non-`const` member functions on a `const` object).

For logical operators, everyone expects `int` at worst, and `bool` at best

More on Return Value as `const`

Consider `func(a + b)`

- `a + b` will be automatically stored as a const because it is a temporary - so making the return value `const` may seem redundant
- Also, you may want to do `(a + b).func2()`
  - Now, only a `const` function would be executed if the return value is const
- This is actually the correct thing to do - why?
  - `(a + b)` isn't explicitly stored anywhere - so this prevents you from storing potentially valuable information on an object that will likely be lost
  - For example

```
(a + b).func2(); // increment the result by 1

(a + b).func2(); // increment the result by 1
```

What would be the end result?

Return Value Optimization

```
return Integer(left.i + right.i);
```

- This is NOT a function call to a constructor (we have seen this format before in aggregate init)
- This actually means, make a temporary `Integer` object and return it
- This **different** from

```
Integer tmp(left.i + right.i);

return tmp;
```

  - `tmp` object is created using its constructor -> copy-constructor copies its value to where the return value is stored -> destructor is called for `tmp`
- This is less efficient than the first method
  - Compiler directly creates the object into the return value location (i.e., 1 constructor call, no copy-constructor, no destructor)