

Operator Overloading

Wagner Truppel
Lecturer, Dept. of Computer Science & Engineering
UC Riverside

wagner@cs.ucr.edu
<http://www.cs.ucr.edu/~wagner>

<http://www.cs.ucr.edu/cs12>

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 1

Operator Overloading

- **Overloading**: having two or more functions (or operators) with the same name but with different *signatures*.
- What's the **signature** of a function or operator? It's made of the function's **name** plus the **number, types, and order** of its arguments.

```
int f(int a, char b);  
int f(int a);  
int f();
```
- Note that **int f()** and **char f()** are **not** overloaded versions of the same function. The return type is **not** part of the signature. The compiler will complain!

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 2

Operator Overloading

- The most commonly overloaded functions in a class are its **constructors** and some **operators**. But you may overload other functions too□
- Why is it useful to overload operators ?
 - ◆ Because we can then use our classes in a more natural fashion. Example:

```
Money price(50); // a $50 item  
Money tax = 0.08 * price; // 8% tax  
Money total = price + tax; // amount due
```

instead of having to manipulate the class member variables every time.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 3

3 ways to overload operators

- *Not* as member functions of the class
 - ◆ Somewhat inefficient because they won't have access to the class' private member variables and, so, must use the accessor and mutator methods.
 - ◆ Have as many arguments as required by their nature (unary, binary, etc)
- As member functions of the class
 - ◆ Have access to private member variables
 - ◆ Have *one less* argument than required by their nature
 - ◆ Have an implicit argument, the object on which they're being invoked.
- As *friends* of the class (more later)

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 4

Not as class members

```

class Money
{
    private:
        int dollars;
        int cents;
    public:
        Money();
        Money(int dollarAmt);
        Money(int dollarAmt, int centsAmt);
        int getDollars();
        int getCents();
};

const Money operator -(const Money &amt)
{ return Money(-amt.getDollars(), -amt.getCents()); }
    
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 5

As class members

```

class Money
{
    private:
        int dollars;
        int cents;
    public:
        Money();
        Money(int dollarAmt);
        Money(int dollarAmt, int centsAmt);
        int getDollars();
        int getCents();
        const Money operator -() const;
};

const Money Money::operator -() const
{ return Money(-dollars, -cents); }
    
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 6

Friend fs and ops

- Is this legal?
 - ◆ Money bill(10);
 - ◆ Money cost = bill + 2;
- Well... here's what the compiler does:
 - ◆ Is there an overloaded version of + taking one argument of type Money and another of type int? If so, no problem
 - ◆ Is there a constructor taking a single int argument? If so, no problem
 - ◆ If none of those happened, then the compiler complains
- How about this?
 - ◆ Money cost = 2 + bill;

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 7

Friend fs and ops

- How about this?
 - ◆ Money cost = 2 + bill;
- It may or may not be legal...
- If + is overloaded outside the class, it's legal; if + is overloaded as a member op, it's not legal
- That's because if + is overloaded as a member op, then its first argument is implicitly an object of the class. Here, though, 2 is **not** an object of class Money, so the compiler complains.
- Confusing, huh? (That's C++ for ya...)

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 8

Friend fs and ops

- So, it pays to have ops be overloaded outside the class. But, then, they don't have access to the private member variables, which is inefficient.
- What's the fix? Overload operators as *friends* of the class!
- **Friend** functions: **not** member functions but they **do** have access to the private member variables of the class.
- Friends are **always** public !
- Some compilers get confused by friends
- Who ever said that C++ is friendly ? :)

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 9

As friends

```

class Money
{
    private:
        int dollars;
        int cents;
    public:
        Money();
        Money(int dollarAmt, int centsAmt);
        int getDollars();
        int getCents();
        friend const Money operator -(const Money& amt);
};

const Money operator -(const Money& amt)
{ return Money(-amt.dollars, -amt.cents); }
    
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 10

References

```

2; // the value 2
int x; // the variable named x
int * p = &x; // p is a pointer to x ("the address of x")
int & y = x; // a reference to x

x = 2; // sets x to 2...
*p = 2; // also sets x to 2...
y = 2; // also sets x to 2 !
    
```

- A **reference** is like **another name** for a variable, like a *nickname*
- A **reference** is *actually* a **constant pointer that is automatically dereferenced**.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 11

References

```

int x; // the variable named x
int & y = x; // a reference to x
    
```

- A **reference** is like **another name** for a variable, like a *nickname*
- A **reference** is *actually* a **constant pointer that is automatically dereferenced**.
- It *is* a pointer... though it doesn't look like it behaves as one.
- It's a **constant** pointer... once you've assigned it, you cannot re-assign it.
- It's **automatically dereferenced**... you don't need to use the * operator to get to the variable it references.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 12

References and functions

- References are most useful in the context of functions

```
int f(int x) // dumb function...
{ return x; }
```
- What's $f(z)$? It's the **value** of z ...
- You cannot, for example, write $f(z) = 3$; $f(z)$ is *not* an L-value.

```
int &g(int &x) // same dumb function?
{ return x; }
```
- What's $g(z)$? It's as if you had returned the variable z itself

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 13

References and functions

- Since $f(z)$ is the value of z , it does not make sense to try, say, $f(z) = 3$;
- However, since $g(z)$ is a reference to the variable z , it's ok to write $g(z) = 3$; The result, in this example, is setting the value of z to 3.
- How about this?

```
int &h(int x) // another dumb function
{ return x; }
```
- What's $h(z)$ now?
- It's a reference to the variable x , **not** to the variable z .
- But x is local and disappears after the function exits, so... trouble!

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 14

References and functions

- Bottom line is: there are important differences between the "value" of a variable, the "name" of a variable, and the "address" of a variable.
- Date getDate(); // returns the actual obj
- Date & getDate(); // returns a reference
- Date * getDate(); // returns a pointer
- You need to think very carefully before you decide what the input parameters should be and also what the return result should be

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 15

Overloading << & >>

- In the expression `cout << x;` the `<<` is a binary operator; its arguments are `cout` and `x`.
- It takes `cout` and `x`, appends `x` to `cout`, then returns a **reference to `cout`**, so that we can chain more invocations of `<<`:
`((cout << x) << " ") << y`
- We don't need the `()`'s, but that's what's happening

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 16

Overloading << & >>

- So, if we want to overload `<<`, we cannot do it inside the class because overloaded member operations assume the first argument to be the object we're invoking the operator on.
- `<<` **must** be overloaded outside the class. For efficiency reasons, make it a *friend* of the class.
- But, wait... why do we want to overload `<<` for a given class to begin with?

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 17

Overloading << & >>

- But, wait... why do we want to overload `<<` for a given class to begin with?
- So that we can output an object's contents more easily, simply by saying `cout << theObject;`
- Example: suppose we want to print the `Money` object `Money(3, 17)` as `$3.17`. We can overload `<<` so that `cout << Money(3, 17);` does just that.
- Read the details in the book... the rest is similar to what we've seen already.
- Similar comments apply to `>>`.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 18

Overloading =

- What really happens when you write
`Money m1(10, 0);`
`Money m2 = m1;`
- = is the **assignment** operator.
- It's a **binary** operator.
- If you don't overload it, the compiler gives you a default version, which copies the **values** of all the variables from one object to the other.
- That's fine if none of your objects have member variables of pointer type.
- If your class uses pointers as member variables, you **should** overload =.
- More on this later.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 19

Overloading prefix and postfix operators

- Example: `++x` and `x++`
- They're both unary operators, and they have the same signature. How can you overload them both?
- You need to pass a dummy variable of type `int` to the postfix version (How clunky is *that*, huh ??)

```
Money & operator ++(); // prefix version  
Money operator ++(int unused); // postfix version
```

- Note: no `const` !
- Note different return types !

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 20

Back to Dynamic Memory Allocation

- Recall that we talked about pointer variables, and their connection with dynamically allocated arrays and structs
- Recall that if you have a pointer to a structure, you can access the structs member variables in one of two ways:
 - `struct Point { int x; int y; };`
 - `Point * p = new Point;`
 - `(*p).x = 2; (*p).y = 3; // set (x,y) to (2,3)`
 - `p -> x = 2; p -> y = 3; // set (x,y) to (2,3)`

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 21

Back to Dynamic Memory Allocation

- You can do the same with a class:
`Money * mptr = new Money(4, 50);`
`cout << mptr -> getDollars(); // prints 4`
- Wait... we've never seen **new** used with classes before. What's the difference between **Money(5, 40)** and **new Money(5, 40)** ?
- The difference is somewhat subtle...
 - ◆ An object allocated **with new** exists until you actually **delete** it
 - ◆ An object allocated **without new** exists only until the **end of the scope** where it's allocated ("the next closing brace")

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 22

this

- Suppose you're writing a member function, say, **g()**, for a class, say, **Money**. How's **g** invoked?
`Money m(4, 50);`
`m.g(); // this invokes g() on the object m`
- Now... how do you refer to the object **m** inside **g()** since, at the time that you're writing **g()**, you don't know about **m** ?
- You use **this**: a **pointer** to the object on which the function or operation is invoked.
- **this** is a member variable defined for every object; it's a **pointer** to the object and **cannot** be changed.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 23

this

- Why would anyone want to use such a creature?
- Well, consider this... (no pun intended)
- You're writing a member function that does something to the calling object and, in the end, you want to return the calling object itself. How would you do it if you did not have some way of referring to it?
- Case in point: overloading the assignment operator

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 24

Overloading =

- If you don't do it yourself, the compiler creates a default version for you
- Why would you want or need to overload = ?
- The default version just copies all the member variables from one object to another
- That's fine if your member variables aren't pointers
- But...

before
m2 = m1;

Money m1	Money m2
- int dollars = 5	- int dollars = 5
- int cents = 40	- int cents = 40

after
m2 = m1;

Money m1	Money m2
- int dollars = 5	- int dollars = 5
- int cents = 40	- int cents = 40

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 25

Overloading =

before
b2 = b1;

an array of Account objects

↑

Bank b1

- Account acct[A]
- int bankID = 2

NULL

↑

Bank b2

- Account acct[A]
- int bankID

after
b2 = b1;

an array of Account objects

↑

Bank b1

- Account acct[A]
- int bankID = 2

an array of Account objects

↑

Bank b2

- Account acct[A]
- int bankID = 2

- The **default** version just copies all the member variables from one object to another
- That's fine if your member variables aren't pointers
- But if you have pointers, you get more than one object pointing to the same area of memory.
- The **pointers** get copied, but *not what they point to!*
- This is called a **shallow copy**.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 26

Overloading =

before
b2 = b1;

an array of Account objects

↑

Bank b1

- Account acct[A]
- int bankID = 2

NULL

↑

Bank b2

- Account acct[A]
- int bankID

after
b2 = b1;

an array of Account objects

↑

Bank b1

- Account acct[A]
- int bankID = 2

an array of Account objects

↑

Bank b2

- Account acct[A]
- int bankID = 2

- Often times, what you want is a **deep copy**.
- In a deep copy, **everything** gets copied from one object to another.
- The way to get a deep copy is to overload the assignment operator.
- You can **only** overload the assignment operator as a **member operator**.
- It should return a reference to the object on the LHS of =. Use **this** for that (another pun).

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 27

Overloading =

Correct way of overloading =:

```
Class & Class::operator = (const Class & rhs)
{
    if (this != &rhs)
    {
        // copy rhs' member variables into 'this'
        // according to your needs (shallow, deep,
        // or a mixture of both)
    }

    return *this; // note the dereferencing of 'this'
}
```

- The test prevents problems when you try assignment an object to itself.
- The book has more examples.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 28

Copy Constructor

- Typically, if you need to overload =, you also need a **copy constructor**.
- It's a constructor like any other, except that it takes only one argument, an object of its own class:
Foo(const Foo & fooObj)
- Note: call-by-constant-reference
- Should create a complete and independent (deep) copy of its argument.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 29

Copy Constructor

- Can be used like any other constructor.
- But is also invoked automatically in certain occasions:
 - ◆ When a function returns a value of the class type : Foo f()
 - ◆ When a function takes a class type parameter 'by value' : f(Foo fooObj)
- You get one if you don't define one yourself, just like with the = operator.
- Default performs a *shallow* copy.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 30

Destructors

- Recall that dynamically allocated variables need to be disposed of properly.
- If your class stores dynamically allocated variables (dynamic arrays, other objects, variables created with new), you need to clean up.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 31

Destructors

- If you create a local variable of a class type (a local object) then, when that block of code ends, the object “goes out of scope.”
- The system automatically invokes a certain member function of the class before the object goes out of scope, to give you a chance to clean up.
- That member function is called the **class destructor**.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 32

Destructors

- They have the same name as the class, but preceded by a tilde.
- They have no arguments.
- They have no return type, not even void.
- There can be only one.
- You cannot overload it.
- `~Foo()` // destructor for class Foo

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 33

The Big Three

- Typically, if your class requires one of these:
 - ★ Overloaded =
 - ★ Copy constructor
 - ★ Destructor

You require them all so, if you define one, you should define all three.

- The compiler will generate default versions of them for you, but those default versions may not do the right thing.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 34
