COP3503

Operator Overloading

What Are Operators?

- Things you've already used before: `+ - / * += < != etc.…`
- What are they, **really**?
- Functions with **operator** in the name:

You can also overload operators as regular, non-class functions, but we won't cover that.

```
void operator*(double y);              // Function name: operator*
bool operator<(Vector3& rhs);          // Function name: operator<
SomeObject& operator[](int index);     // Function name: operator[]
Object& operator=(const Object& m);    // Function name: operator=
```

An unusual name, but still just a name.

- These functions are used to customize the way you work with classes.

Overloading Functions

- **Function overloading** is what allows you to create multiple versions of a function, each with different signatures (different parameter lists).

```
void Foo(int x, float y);
void Foo(string message, int x);
void Foo(string message, int x, float y);
// etc...
```

- Overloading an **operator** is the similar concept.

Example

```
Vector3 x, someDirection(1, 0, 0);
x = someDirection; // What's happening here?
```

This is assigning `someDirection` to `x` by using the copy assignment operator.

Whether you created it as part of the Big Three, or it's implicitly created for you.

- What's really happening is just a function call:

```
x.operator=(someDirection); // Just a function with an odd name...
```

P.S. Don't ever write code like this. If you do, your instructor will cry.

…and your coworkers. Basically, anyone who sees your code.

- Operators can be overloaded as a shortcut and an (arguably) cleaner-looking alternative to the previous line.

Operators Are Shorthand for Functions

```
Vector3 direction(2, 4, 2);
Vector3 up(0, 1, 0);
```

Traditional Function

```
Vector3 result = direction.Add(up);

// Double the length of the vector
direction.Scale(2.0f);

// Add multiple vectors
Vector3 a, b, c;
/* assume initialization */
result = a.Add(b.Add(c)); // Ew
```

Operator Function

```
Vector3 result = direction + up;

// Double the length of the vector
direction *= 2.0f;


Vector3 a, b, c;
// Much nicer!
result = a + b + c;
```

Using the return of a function as a parameter to another function isn't inherently bad. It might make your code a bit harder to read. (A subjective thing)

What Operators Can You Overload?

Arithmetic Operators	Relational Operators	Assignment and Compound Operators
+	< and <=	=
-	> and >=	+=
*	==	-=
/	!=	*=
		/=

- And many, many more…

Want to treat a class like an array? There's also the subscript operator (the brackets operator): **[ ]**

Overloaded Operators Should Match Expectations

```
// Assignment
int a, b = 2, c = 5;
a = b + c; // combine b and c into a new integer

// You should be able to catch the results
result = thingA + thingB;
thingA = thingA + thingB; // thingA is changed by assignment

// Examples: The + operator should NOT change things
int a = b + c;   // Don't change b or c
int x = 5 + 12;  // 5 stays 5, 12 stays 12
b + x;           // Result: Sum of b and x is discarded
```

This code, applied to primitive types, should have no effect (and your compiler may even ignore it in the end).

For a class, the same behavior should be followed.

Bad Example

```
class Vector3
{
    float x, y, z;
public:
    Vector3()
    { x = y = z = 0; }

    void operator+(float value)
    {
        x += value;
        y += value;
        z += value;
    }
};
```

This modifies **obj**– not good!

Modifying **this** is atypical behavior for this type of operator; don't surprise other people using your code!

```
// Elsewhere...
Vector3 obj;
//obj.operator+(4.2) (don't do this)
obj + 4.2;

Vector3 obj2 = obj + 25;
```

Lacking any other info... this implies **no change** to **obj**.

Bad Example

```
class Vector3
{
    float x, y, z;
public:
    Vector3()
    { x = y = z = 0; }

    void operator+(float value)
    {
        x += value;
        y += value;
        z += value;
    }
};
```

```
// Elsewhere...
Vector3 obj;
//obj.operator+(4.2) (don't do this)
obj + 4.2;
```

```
Vector3 obj2 = obj + 25;
```

This will not compile; the + operator returns **void**, which won't work with a copy constructor (or any function).

```
Vector3 obj2 = void; // error
```

You can't assign **void** to anything!

Good Example

```
class Vector3
{
    float x, y, z;
public:
    Vector3()
    { x = y = z = 0; }
    Vector3 operator+(float value)
    {
        Vector3 newVec;
        newVec.x = x + value;
        newVec.y = y + value;
        newVec.z = z + value;
        return newVec;
    }
};
```

Why create a new object and return it by value? Isn't that slower than passing by reference?

Yes, it is, but that's how it ought to be: Create something new from two other things.

```
// Elsewhere...
Vector3 obj;
obj + 4.2;

Vector3 obj2 = obj + 25;
```

This will not change the invoking object (**obj**).

This works as expected, creating a new object (in the operator) and then copying it into another via the copy constructor.

Assignment Equals Change

```
class Vector3
{
    float x, y, z;
public:
    Vector3()
    { x = y = z = 0; }
    Vector3& operator+=(float value)
    {
        x += value; // Change “this”
        y += value;
        z += value;
        return *this;
    }
};
```

```
// Elsewhere...
Vector3 obj;
obj += 4.2;
```

Anyone reading this would expect **obj** to change on this line…

+=, -=, *=, /=

It is expected these operators **will** change the invoking object.

Any unusual behaviors should **not** be operators; it makes your code harder to understand.

What if Assignment Didn't Equal Change?

```
class Vector3
{
    float x, y, z;
public:
    Vector3()
    { x = y = z = 0; }
    Vector3 operator+=(float value)
    {
        Vector3 newVec;
        newVec.x = this->x + value;
        newVec.y = this->y + value;
        newVec.z = this->z + value;
        return newVec;
    }
};
```

This does **not** change the invoking object.

What if this didn't work:

```
int x = 10;
x += 5; // Nope, won't change x!

x = x += 5; // Changes x!
```

The syntax for this is… not good.

```
// Elsewhere.
Vector3 obj;
obj += 4.2;
```

This line now does something other than what is expected… unpredictability is bad in programming.

Be cautious when customizing operators. Just because you **can** do something…doesn't mean you **should**, or that it's a good idea!

Overloading an Operator Multiple Times

```
class Vector3
{
    float x, y, z;
public:
    Vector3();

    // Operator overloading... overloading?
    Vector3& operator+=(float value);
    Vector3& operator+=(const Vector3& value);
};

Vector3 a, b;
// Two separate use-cases
a += 5;
b += a;
```

There's nothing wrong with this. Just be sure each operator does what it advertises (change the object, in this case).

Alternatives to Operators

- Use clearly-defined function names to indicate behaviors.
- No one should have to guess what an operator really does behind the scenes.
- You **can** create t

```
BankAccount a, b;
a += b; // Uhh, what? Adding two bank accounts?

// Clarity about WHAT is happening (the HOW is unimportant)
a.MergeWithAccount(b);
```

Side note: In the grand scheme of things, some other class would probably be a better idea to manage the merging of two BankAccounts

```
class BankAccount
{
    float balance;
public:
    BankAccount& oper
    { /*Combine the t

    // A better choice, behavior is more apparent
    void MergeWithAccount(BankAccount& otherAccount)
    { /*Combine the two account balances*/ }
};
```

Other Things You Can Do With Operators

- Make them private, so no one can use them
  - This also applies to things like constructors, copy constructors.

```
class SomeClass
{
    // Private operator, cannot be used outside of this class
    SomeClass& operator=(const SomeClass& s) {/*definition omitted*/}
};

int main()
{
    SomeClass a, b;
    a = b; // Compiler error, cannot access private function

    return 0;
}
```

Why do this? Maybe you never intend for an object to be copied (behind the scenes it accesses some resources that shouldn't be copied?)

No Limit to the Number of Overloaded Operators

```
class Matrix4x4
{
    float data[4][4];
public:
    Matrix4x4 operator-(const Matrix4x4& rhs);
    Matrix4x4 operator+(const Matrix4x4& rhs);
    Matrix4x4 operator*(const Matrix4x4& rhs);
    Matrix4x4& operator-=(const Matrix4x4& rhs);
    Matrix4x4& operator+=(const Matrix4x4& rhs);
    Matrix4x4& operator*=(const Matrix4x4& rhs);
    // etc...
};
```

rhs == Right Hand Side

With overloaded operators, what's on the right side of the operator gets passed to the function.

```
Matrix4x4 a, b;
a += b; // b == rhs
a.operator+=(b);
```

- At some point you'll hit a "soft" limit, and you don't need any more.
- The language doesn't set a limit—you can implement as many as you want.
- Math-related classes might implement (and use!) the most.

Relational Operators

- Is this **equal** to that?
- Is this **not equal** to that?
- Is this object **greater than**, or **less than** that?

```
bool operator==(const SomeClass& thingToCompare); //
bool operator!=(const SomeClass& thingToCompare); //
bool operator>(const SomeClass& thingToCompare);
bool operator<(const SomeClass& thingToCompare);
```

Exactly **how** you implement these will depend on the class and its member variables.

Example: operator==

```
class Hero
{
    string _name;
    int _hitpoints;
    int _maximumHitpoints;
public:
    bool operator==(const Hero& otherHero);
};

bool Hero::operator==(const Hero& otherHero)
{
    return (_name == otherHero._name &&
    _hitpoints == otherHero._hitpoints &&
    _maximumHitpoints == otherHero._maximumHitpoints)
}
```

If everything is the same, return true. Otherwise, return false!

```
vector<Hero> heroes;
Hero someHero;
for (int i = 0; i < heroes.size(); i++)
{
    if (someHero == heroes[i])
    {
        cout << "Found hero at index: " << i;
        break;
    }
}
```

An operator lets us write code like this, to help with searches.

Example: operator==

```
class Hero
{
    string _name;
    int _hitpoints;
    int _maximumHitpoints;
public:
    bool operator==(const Hero& otherHero);
};

bool Hero::operator==(const Hero& otherHero)
{
    // Check if anything is NOT equal... if so, stop checking!
    if (_name != otherHero._name) return false;
    if (_hitpoints != otherHero._hitpoints) return false;
    if (_maximumHitpoints != otherHero._maximumHitpoints) return false;

    // Made it here? No inequalities anywhere? Must be the same!
    return true;
}
```

This is just one example. If a class had **lots** of member variables, you might only care about a subset of them.

Maybe **just** the name is enough to be considered equal. Maybe in your program you only care about heroes with the same number of hitpoints.

You can customize your operators to work any way that you want.

Example: operator==

```
class Hero
{
    string _name;
    int _hitpoints;
    int _maximumHitpoints;
public:
    bool operator==(const Hero& otherHero);
}

bool Hero::operator==(const Hero& otherHero)
{
    return (_hitpoints == otherHero._hitpoints);
}
```

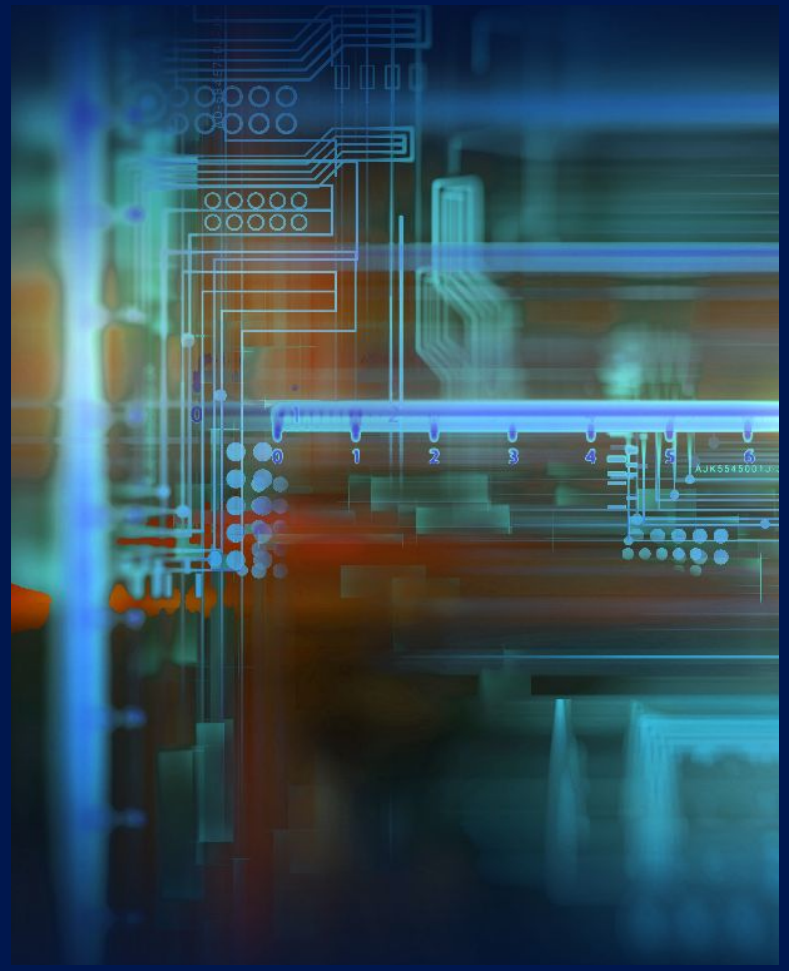
Not inherently wrong. But might not be right in all programs.

The same goes for the other operators:
operator!=
operator>
operator<

Recap

- Operators are functions, and functions can be overloaded
- Operators can create code shortcuts.
- You won't overload operators all the time (nor should you!).
  - Not all languages support overloading operators.
- In some cases you **must**, for correct functionality.
  - Assignment operators, when part of the "Big Three"
- If it make logical sense…
  - **And** it maintains or improves the readability of your code
  - Copy assignment operator: **member-to-member copy**
- Don't do it "just because".

Conclusion



Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Thank you for watching.