

Lecture 8.5: Operator Overload Overflow

Bart Iver van Blokland

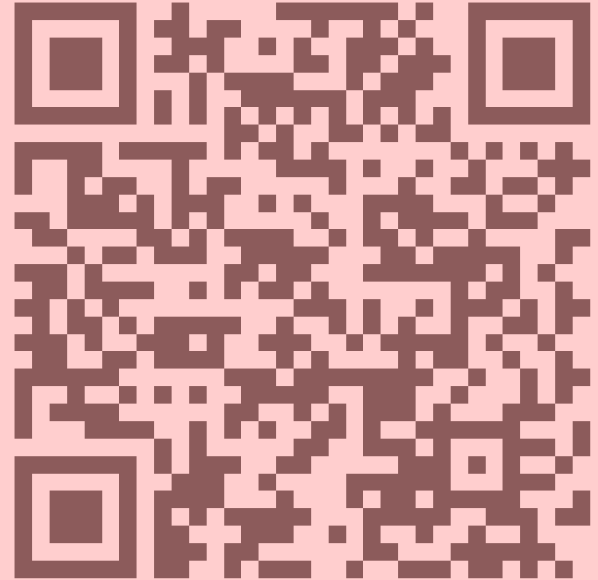
PSA: Reminder: halfway evaluation

- Thanks to everyone that has responded thus far!
- Should only take a few minutes to give some ratings, but text responses tend to be most useful for making improvements



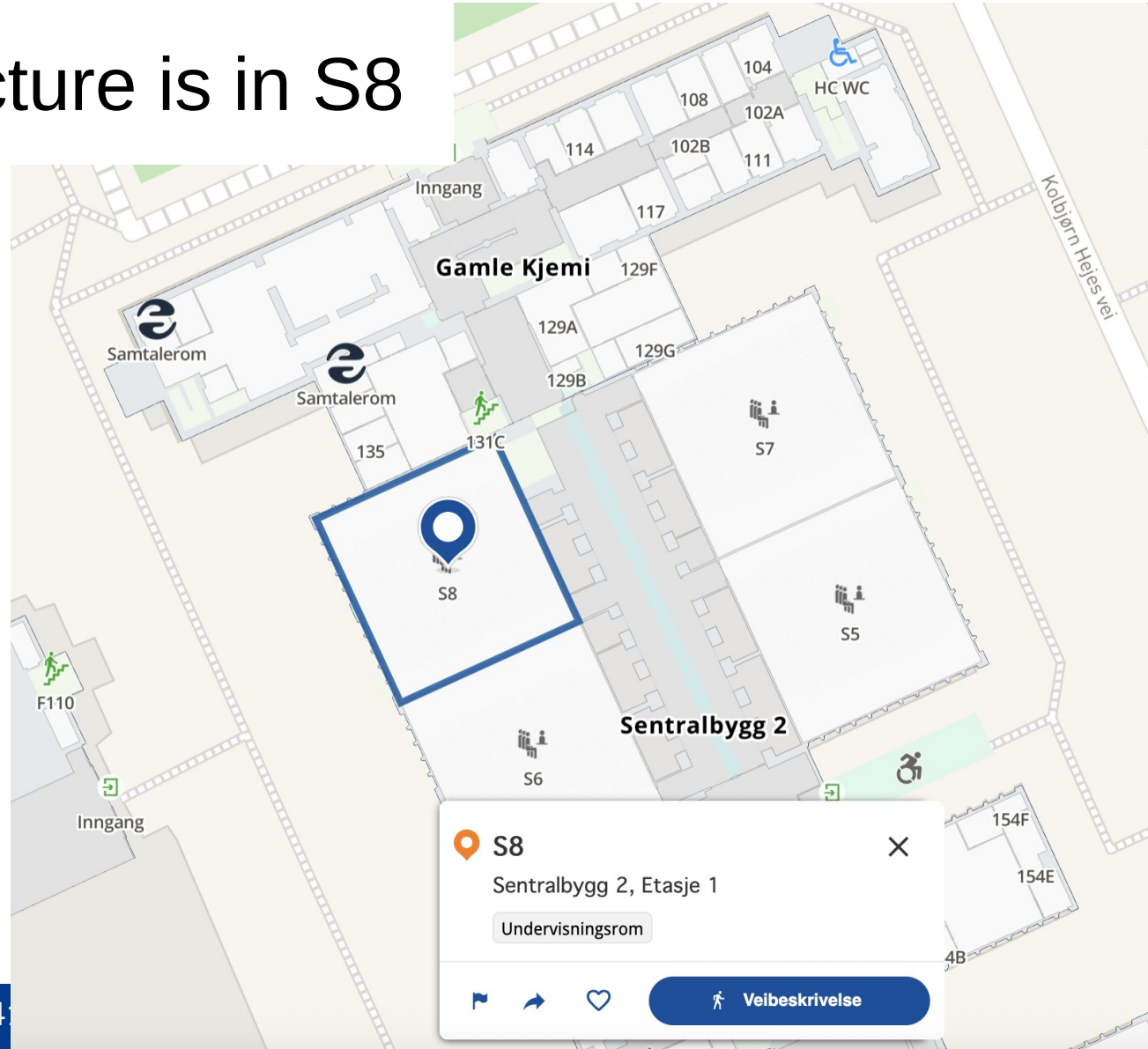
PSA: Reminder: halfway evaluation

- Thanks to everyone that has responded thus far!
- Should only take a few minutes to give some ratings, but text responses tend to be most useful for making improvements



PSA: Thursday's lecture is in S8

- We were asked if we *pleasepleasepleaseplease* could move because they wanted to do an all-day event in R1.
- Next week's lectures are back in R1 as usual.



Today

OPERATOR OVERLOADING



Demonstration: operator overloading


```

4   struct FloatingPoint {
5       double x = 0;
6       double y = 0;
7   };
8
9   int main() {
10      FloatingPoint a {4, 5};
11      FloatingPoint b {6, 7};
12
13      if(a == b) {
14
15      }
16

```

TERMINAL

...



Build Executable - Task



+ v



...



INFO: autodetecting backend as ninja ...

../main.cpp:14:10: **error:** invalid operands to binary expression on ('FloatingPoint' and 'FloatingPoint')

14 | if(a == b) {

Operator overloading

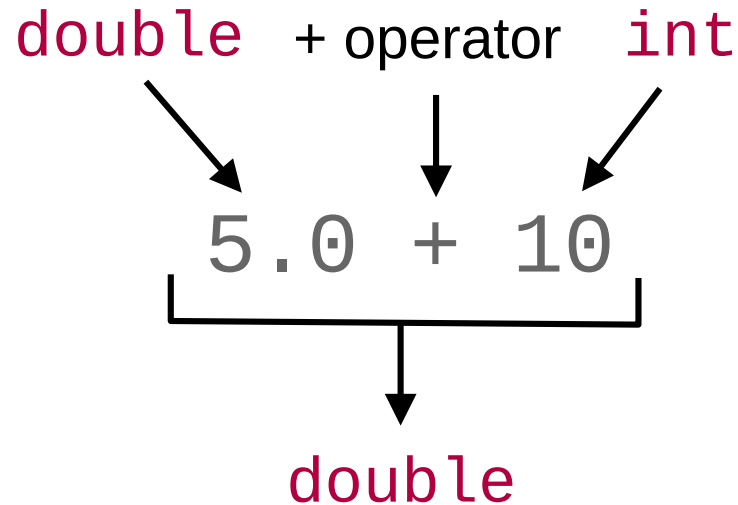
```
../main.cpp:14:10: error: invalid operands to binary expression  
('FloatingPoint' and 'FloatingPoint')  
14 |         if(a == b) {  
    |             ~ ^ ~
```

Operands: the values
used with the operator.
Here: a and b

Binary expression: a computation
done with two values

Operator overloading

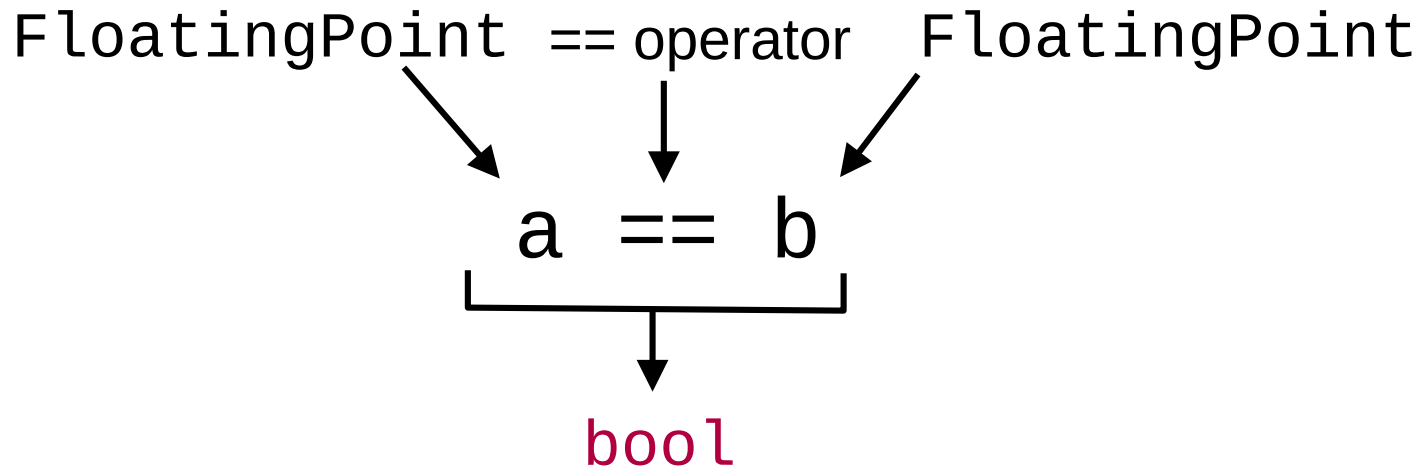
You can think of an operator as a function that takes two parameters and returns another value



Operator overloading

```
../main.cpp:14:10: error: invalid operands to binary expression  
('FloatingPoint' and 'FloatingPoint')
```

```
14 |         if(a == b) {  
    |             ~ ^ ~
```



Operator overloading: syntax

value1 > value2

```
dataType operator>(dataType lhs, dataType rhs) {  
    // Implement the operator here  
}
```

The data type of the result of the operator
(for comparisons like > that is usually **bool**)

Each dataType can be a different type!

Note: the parameter variables are called lhs and rhs here for «left hand side» and «right hand side». This is used consistently throughout the slides.

Operators: There are many!

```
type operator+ (type lhs, type rhs);  
type operator- (type lhs, type rhs);  
type operator* (type lhs, type rhs);  
type operator/ (type lhs, type rhs);  
type operator% (type lhs, type rhs);  
type operator^ (type lhs, type rhs);  
type operator& (type lhs, type rhs);  
type operator| (type lhs, type rhs);  
type operator, (type lhs, type rhs);  
type operator>> (type lhs, type rhs);  
type operator<< (type lhs, type rhs);  
type operator~ (type rhs);  
type operator! (type rhs);  
type operator++ (type lhs);  
type operator-- (type lhs);
```

```
type operator== (type lhs, type rhs);  
type operator!= (type lhs, type rhs);  
type operator&& (type lhs, type rhs);  
type operator|| (type lhs, type rhs);  
type operator< (type lhs, type rhs);  
type operator> (type lhs, type rhs);  
type operator<= (type lhs, type rhs);  
type operator>= (type lhs, type rhs);
```

For all of these, type can be replaced with **any** data type!

(exception: operators with data types that already exist, like `int + int`)

Task: Define an operator

- This code snippet does not compile. Define the missing operator.

```
int main() {  
    std::string message = "The number is: ";  
    std::string combined = message + 5;  
    return 0;  
}
```

Expected result: 'combined' should contain «The number is: 5»

The general operator template, for reference:

value1 > value2



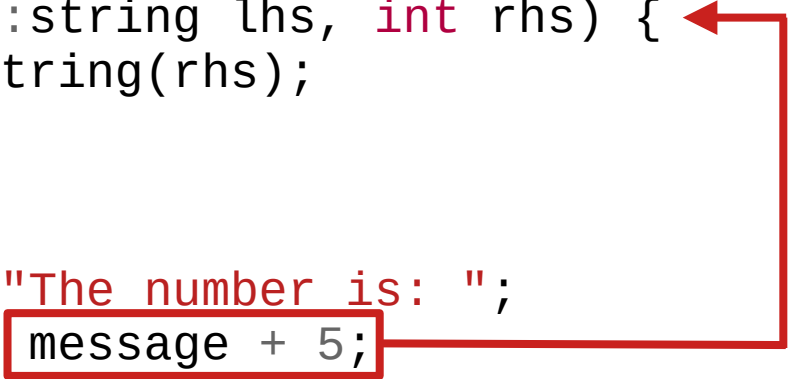
```
dataType operator>(dataType lhs, dataType rhs) {  
    // Implement the operator here  
}
```

Operators are functions

- Operators are normal functions
 - The name must be **operator** with the desired operator appended to it (e.g. **operator*** or **operator!=**)
 - Using the operator calls the operator function with the operands as parameters

```
std::string operator+(std::string lhs, int rhs) {  
    return lhs + std::to_string(rhs);  
}
```

```
int main() {  
    std::string message = "The number is: ";  
    std::string combined = message + 5;  
    return 0;  
}
```

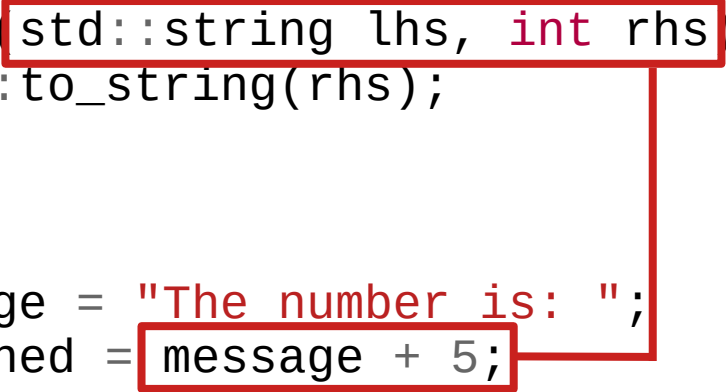


Using the operator calls the operator function!

Operators: Order matters

- The parameter order matters!
 - The example below will not compile, as the operator is only defined for «int + string», not «string + int»

```
std::string operator+(std::string lhs, int rhs) {  
    return lhs + std::to_string(rhs);  
}  
  
int main() {  
    std::string message = "The number is: ";  
    std::string combined = message + 5;  
    return 0;  
}
```



Operator overloading: notes

- Just like any other function, parameters to overloaded operators can be taken in as any data type, including (const) references
- And the same goes for return types. They can be anything you want
 - But you probably should return sensible types, like a bool from a < operator

Today

- Operator overloading: basics
- **Operators as member functions**

Operators: Can be member functions


- When declaring an operand as a member function, the containing class becomes the operator's first parameter

```
// Point.h
struct Point {
    double x = 0;
    double y = 0;

    Point operator+ (Point rhs);
};

// Point.cpp
Point Point::operator+ (Point rhs) {
    return {x + rhs.x, y + rhs.y};
}
```

```
int main() {
    Point a {1, 2};
    Point b {3, 4};
    Point c = a + b;
}
```



operator+() is called for a,
with b as its parameter.

The result is stored in c.

Operators: Can be member functions

- When declaring an operand as a member function, the containing class becomes the operator's first parameter

```
// Point.h
```

```
struct Point {  
    double x = 0;  
    double y = 0;  
  
    Point operator+  
};
```

```
// Point.cpp
```

```
Point Point::operator+ (Point rhs) {  
    return {x + rhs.x, y + rhs.y};  
}
```

Note: the operator is a regular method, so you need to add Point:: in front when you define it.

```
int main() {  
    Point a {1, 2};  
    Point b {3, 4};  
    Point c = a + b;
```

operator+() is called for a, with b as its parameter.

The result is stored in c.

Operators: There are even more!

```
struct dataType {  
    dataType operator[] (dataType rhs) {}  
    dataType operator() (dataType rhs) {}  
    dataType operator= (dataType rhs) {}  
    dataType operator+= (dataType rhs) {}  
    dataType operator-= (dataType rhs) {}  
    dataType operator*= (dataType rhs) {}  
    dataType operator/= (dataType rhs) {}  
    dataType operator%= (dataType rhs) {}  
    dataType operator^= (dataType rhs) {}  
    dataType operator&= (dataType rhs) {}  
    dataType operator|= (dataType rhs) {}  
    dataType operator>>=(dataType rhs) {}  
    dataType operator<<=(dataType rhs) {}  
};
```

.. But these can **only** be declared as a member function!

As before, dataType can be replaced with any other data type!

Operators: Can be member functions

- Binary operators can be either defined as a method or standalone function

```
struct Point {  
    double x = 0;  
    double y = 0;
```

```
    Point operator+ (Point rhs) {  
        return {x + rhs.x, y + rhs.y};  
    }
```

```
};
```

```
Point operator+(Point lhs, Point rhs) {  
    Point sum {lhs.x + rhs.x, lhs.y + rhs.y};  
    return sum;  
}
```

These are equivalent!

e.g. both allow you to
write `a + b`

Today

- Operator overloading: basics
- Operators as member functions
- **Where are operators used?**

Operator overloading: examples in STL

- `std::filesystem::path`:

```
std::filesystem::path dir = "directory";  
dir /= "directory2";  
std::filesystem::path file = dir / "file.txt";
```

- `std::string`

```
std::string message1 = "Hello";  
std::string message2 = "There";  
message1 += " " + message2;
```

Always has been.

**Wait.. Streams are just
objects with a bunch of
overloaded << and >>
operators?**



Stream model

- Implementation: relies on operator overloading!
- Only basic types are supported by `std::ostream` itself

```
class ostream {  
    std::ostream& operator<<(bool rhs);  
    std::ostream& operator<<(short rhs);  
    std::ostream& operator<<(unsigned short rhs);  
    std::ostream& operator<<(int rhs);  
    std::ostream& operator<<(unsigned int rhs);  
    std::ostream& operator<<(long rhs);  
    std::ostream& operator<<(unsigned long rhs);  
    std::ostream& operator<<(long long rhs);  
    std::ostream& operator<<(unsigned long long rhs);  
    std::ostream& operator<<(float rhs);  
    std::ostream& operator<<(double rhs);  
    std::ostream& operator<<(long double rhs);  
    std::ostream& operator<<(const void* rhs);  
};
```

Stream model

```
std::cout << -5 << std::endl;
```

```
class ostream {  
    std::ostream& operator<<(bool rhs);  
    std::ostream& operator<<(short rhs);  
    std::ostream& operator<<(unsigned short rhs);  
    std::ostream& operator<<(int rhs);  
    std::ostream& operator<<(unsigned int rhs);  
    std::ostream& operator<<(long rhs);  
    std::ostream& operator<<(unsigned long rhs);  
    std::ostream& operator<<(long long rhs);  
    std::ostream& operator<<(unsigned long long rhs);  
    std::ostream& operator<<(float rhs);  
    std::ostream& operator<<(double rhs);  
    std::ostream& operator<<(long double rhs);  
    std::ostream& operator<<(const void* rhs);  
};
```

Stream model

```
std::cout << 33.6f << std::endl;
```

```
class ostream {  
    std::ostream& operator<<(bool rhs);  
    std::ostream& operator<<(short rhs);  
    std::ostream& operator<<(unsigned short rhs);  
    std::ostream& operator<<(int rhs);  
    std::ostream& operator<<(unsigned int rhs);  
    std::ostream& operator<<(long rhs);  
    std::ostream& operator<<(unsigned long rhs);  
    std::ostream& operator<<(long long rhs);  
    std::ostream& operator<<(unsigned long long rhs);  
    std::ostream& operator<<(float rhs);  
    std::ostream& operator<<(double rhs);  
    std::ostream& operator<<(long double rhs);  
    std::ostream& operator<<(const void* rhs);  
};
```

Stream model

```
std::cout << 33.6 << std::endl;
```

```
class ostream {  
    std::ostream& operator<<(bool rhs);  
    std::ostream& operator<<(short rhs);  
    std::ostream& operator<<(unsigned short rhs);  
    std::ostream& operator<<(int rhs);  
    std::ostream& operator<<(unsigned int rhs);  
    std::ostream& operator<<(long rhs);  
    std::ostream& operator<<(unsigned long rhs);  
    std::ostream& operator<<(long long rhs);  
    std::ostream& operator<<(unsigned long long rhs);  
    std::ostream& operator<<(float rhs);  
    std::ostream& operator<<(double rhs);  
    std::ostream& operator<<(long double rhs);  
    std::ostream& operator<<(const void* rhs);  
};
```

Stream model

- If anything is missing, we can create our own operator
- It can be used with all output streams (e.g. `std::cout` or `std::ofstream`)!
 - How this works is a topic for a later lecture

```
std::ostream& operator<<(std::ostream& lhs, FloatingPoint rhs) {  
    lhs << "(" << rhs.x << ", " << rhs.y << " )";  
    return lhs;  
}
```

Allows you to do this:

```
FloatingPoint a {4, 5};  
std::cout << a << std::endl;
```


Stream model

- Possible exam questions:
 - Why must the ostream object in the example below be a reference?
 - Why must the ostream object in the example below be returned?

```
std::ostream& operator<<(std::ostream& lhs, FloatingPoint rhs) {  
    lhs << "(" << rhs.x << ", " << rhs.y << " )";  
    return lhs;  
}
```

Allows you to do this:

```
FloatingPoint a {4, 5};  
std::cout << a << std::endl;
```

Stream model

- Returning the ostream object allows you to chain it:

```
std::cout << "From: " << a << " to: " << b << std::endl;
```

Stream model

- Returning the ostream object allows you to chain it:

```
std::cout << "From: " << a << " to: " << b << std::endl;
```

```
std::ostream& operator<<(std::ostream& lhs, std::string rhs) {  
    /* ... */  
    return lhs;  
}
```

Stream model

- Returning the ostream object allows you to chain it:

```
std::cout << a << " to: " << b << std::endl;
```

```
std::ostream& operator<<(std::ostream& lhs, FloatingPoint rhs) {  
    lhs << "(" << rhs.x << ", " << rhs.y << " )";  
    return lhs;  
}
```

Stream model

- Returning the ostream object allows you to chain it:

```
std::cout << " to: " << b << std::endl;
```

```
std::ostream& operator<<(std::ostream& lhs, std::string rhs) {  
    /* ... */  
    return lhs;  
}
```

Stream model

- Returning the ostream object allows you to chain it:

```
std::cout << b << std::endl;
```

```
std::ostream& operator<<(std::ostream& lhs, FloatingPoint rhs) {  
    lhs << "(" << rhs.x << ", " << rhs.y << " )";  
    return lhs;  
}
```

We're technically calling other existing overloads of the << operator here..

Stream model

- Returning the ostream object allows you to chain it:

```
std::cout << std::endl;
```


Returning void results in a compiler error when we try to chain values to print.

```
9 void operator<<(std::ostream& o, FloatingPoint p) {
10     o << "(" << p.x << ", " << p.y << ")";
11 }
12
13 int main() {
14     FloatingPoint a {4, 5};
15     FloatingPoint b {6, 7};
16
17     std::cout << "From " << a << " to " << b << std::endl;
18
19     return 0;
20 }
```

PROBLEMS 2 OUTPUT TERMINAL ... Build Executable - Task (X) + v [icon] [icon] ...

INFO: calculating backend command to run: /opt/homebrew/bin/ninja -C /Users/bart/Documents/NTNU/2023/TDT4102/2026/TDT4102-exercises-proposal_exercise_7/exercise7_new2/grading/handout/build

...
../main.cpp:17:31: **error:** invalid operands to binary expression ('void' and 'const char[5]')

```
17 |     std::cout << "From " << a << " to " << b << std::endl;
    |                                ^
    |                                ~~~~~
```

1 error generated.

Overloading >> for input streams

We can also create operators for `std::cin` and `std::ifstream`.

```
struct Quote {  
    std::string text = "If your code is blurry you may not C#";  
    std::string whoSaidIt = "I";  
};
```

```
std::istream& operator>> (std::istream& lhs, Quote& rhs) {  
    lhs >> rhs.whoSaidIt;  
    std::getline(lhs, rhs.text);  
    return stream;  
}
```

Remember to return
the stream object!

The right hand side
must be a reference!

Spoiler alert!



Printing private members

Printing out private fields of a class is also possible, although it requires declaring the `operator<<` function as a `friend` (explained in a later lecture)

```
class Point {  
    double x = 0;  
    double y = 0;  
    friend std::ostream& operator<< (std::ostream& lhs, Point rhs);  
};  
  
std::ostream& operator<< (std::ostream& lhs, Point rhs) {  
    lhs << "[" << rhs.x << ", " << rhs.y << "];"  
    return lhs;  
}
```

Today

- Operator overloading

Thursday

- Memory addresses
- Pointers
- Manual memory management

Reminder!
**Lecture room is
changed to S8**