Lecture 12: Operators

Today’s Agenda

- Operators
  - (that’s it)

Recap: Objects and Classes

Objects

- Objects encapsulate **data related to a single entity**
  - Define **complex behavior** to work with or process that data: `Student.printEnrollmentRecord(), vector.insert()`
- Objects store **private state** through **instance variables**
  - `Person::name, Vehicle::idNumber`
- Expose **private** state to others through **public instance methods**
  - `Person::getName(), Vehicle::editRegistration(string name)`
  - Allow us to expose state in a way we can control

Time

```
class Time {
    public:
        Time(int seconds, int minutes, int hours);
        int getSeconds();
        int getMinutes();
        int getHours();
        const std::string& toString(); // e.g. 5:32:17
    private:
        int seconds;
        // and other instance vars
}
```

Time

Let’s check whether one time is before another…

```
bool before(const Time& a, const Time& b) {
    if (a.getHours() < b.getHours()) return true;
    if (b.getHours() < a.getHours()) return false;
    // otherwise, we have to compare minutes
    if (a.getMinutes() < b.getMinutes()) return true;
    if (b.getMinutes() < a.getMinutes()) return false;
    // compare seconds...
}
```

🤔 **Question:** Why are the arguments **const**?

Time

```
if (before(a, b)) {  // from somewhere, maybe user input
    cout << "Time a is before Time b." << endl;
}
// this is somewhat hard to read
// unclear whether we’re checking if a is before b
// or if b is before a
```

```
// what if we could just do:
if (a < b) {
    cout << "Time a is before Time b." << endl;
}
```

Operator Overloading

Operator Overloading

Operator overloading tells C++ what it means to use an **operator** on a class we’ve written ourselves.

Operator Overloading

```
+ - * / % ^ & | ~ ! , = < > <= >=
++ -- << >> == != && || += -= *=
/= %= ^= &= |= <<= >>= [] () ->
->* new new[] delete delete[]
```

Operator Overloading

+ - * / % ^ & | ~ ! , **= < > <= >=**

++ -- << >> == != && || += -= *=

/= %= ^= &= |= <<= >>= [] () ->

->* new new[] delete delete[]

Operator Overloading

+ - * / % ^ **&** **|** **~** **!** , = < > <= >=

++ -- << >> == != **&&** **||** += -= *=

/= %= ^= &= |= <<= >>= [] () ->

->* new new[] delete delete[]

Operator Overloading

```
+ - * / % ^ & | ~ ! , = < > <= >=
++ -- << >> == != && || += -= *=
/= %= ^= &= |= <<= >>= [] () ->
->* new new[] delete delete[]
```

Operator Overloading

```
if (before(a, b)) {
    cout << "Time a is before Time b." << endl;
}
```

```
if (a < b) {
    cout << "Time a is before Time b." << endl;
}
```

Two ways to do it:

1) member functions
2) non-member functions

🤯 Wait, what are member functions? 🤯

Member Function

```
Person keith;
keith.enroll(“Stanford”);  // declared inside class Person
```

Non-Member Function

```
Person keith;
enroll(keith, “Stanford”); // declared globally (in main.cpp?)
```

1. Member Functions

Add a function called **`operator@`** to your class:

```
class Time {
    bool operator<(const Time& rhs) const;
    Time operator+(const Time& rhs) const;
    bool operator!() const; // unary, no arguments
}
```

- Call the function with **this** as the left hand side of the expression

1. Member Functions

Add a function called `operator@` to your class:

This...

```
Time a, b;
if (a < b) {
    // do something;
}
```

becomes this

```
Time a, b;
if (a.operator<(b)) {
    // do something;
}
```

1. Member Functions

Add a function called **operator@** to your class:

```
class Time {
    bool operator<(const Time& rhs) const;
    Time operator+(const Time& rhs) const;
    bool operator!() const; // unary, no arguments
}
```

- Call the function on the left hand side of the expression (**this**)
- **Binary operators** (5 **+** 2, “a” **<** “b”): accept the right hand side (**rhs&**) as an argument.
- **Unary operators** (**~**a, **!**b): don’t take any arguments

🕒 Before

```
bool before(const Time& a, const Time& b) {
    if (a.getHours() < b.getHours()) return true;
    if (b.getHours() < a.getHours()) return false;
    // compare minutes, seconds, etc.
}
```

🕚 After

```
class Time {
    bool operator<(const Time& rhs) {
        if (hours < rhs.hours) return true;
        if (rhs.hours < hours) return false;
        // compare minutes, seconds...
    }
}
```

1) we’re in a member function, so `hours` refers to **`this`**`.hours` by default

2) we can access private members like `hours` because we’re in a member function

2. Non-Member Functions

Add a function called **`operator@`** **outside of** your class.

```
bool operator<(const Time& lhs, const Time& rhs);
Time operator+(const Time& lhs, const Time& rhs);
Time& operator+=(Time& lhs, const Time& rhs);
Time operator!(const Time& lhs);
```

Takes **all** of its arguments (both lhs and rhs).

Before

```
bool before(const Time& a, const Time& b) {
    if (a.getHours() < b.getHours()) return true;
    if (b.getHours() < a.getHours()) return false;
    // compare minutes, seconds, etc.
}
```

After

```
bool operator<(const Time& lhs, const Time& rhs) {
    if (lhs.getHours() < rhs.getHours()) return true;
    if (rhs.getHours() < lhs.getHours()) return false;
    // notice: exactly the same except for the function name!
}
```

🤔 Questions? 🤔

Live Code Demo:

Fraction.cpp

Operator Overloading – Non-Member Functions

The STL prefers using **non-member** functions for operator overloading:

1) allows the LHS to be a non-class type (e.g. **`double * Fraction`**)
2) allows us to overload operations with a class we don't control as the LHS

Allow non-member function to access **private** members using **`friend`**:

```
// fraction.h
class Fraction {
    friend Fraction operator*(const Fraction& lhs, const Fraction& rhs);
    friend ostream& operator<<(ostream& out, const Fraction& target);
}
```

Operator Overloading — Non-Member Functions

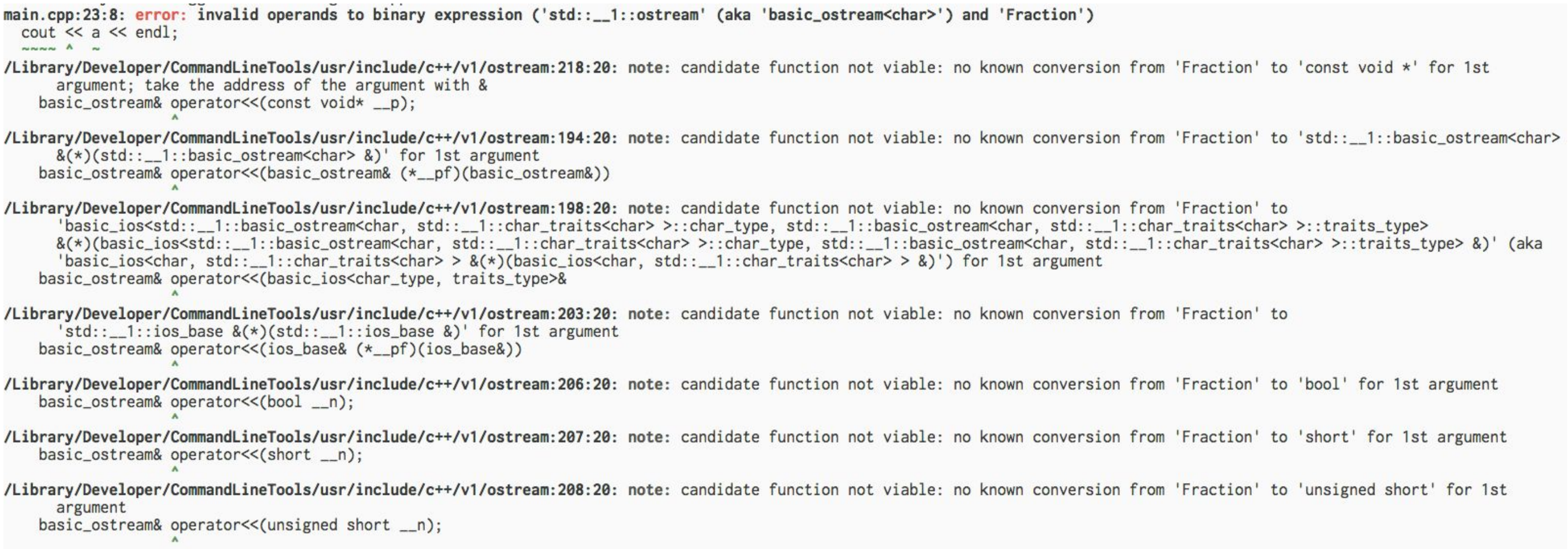
Need access to internal private members? Declare it to be a **friend**:

```
class Person {
    public:
        friend bool operator==(const Person& lhs,
                               const Person& rhs);
    private:
        int secretID;
}

bool operator==(const Person& lhs, const Person& rhs) {
    return (lhs.secretID == rhs.secretID);
}
```

Ever seen this?

```
Fraction a; // our own type
cout << a << endl;
```



```
main.cpp:23:8: error: invalid operands to binary expression ('std::__1::ostream' (aka 'basic_ostream<char>') and 'Fraction')
  cout << a << endl;
  ~~~~ ^ ~
/Library/Developer/CommandLineTools/usr/include/c++/v1/ostream:218:20: note: candidate function not viable: no known conversion from 'Fraction' to 'const void *' for 1st
      argument; take the address of the argument with &
    basic_ostream& operator<<(const void* __p);
                   ^
/Library/Developer/CommandLineTools/usr/include/c++/v1/ostream:194:20: note: candidate function not viable: no known conversion from 'Fraction' to 'std::__1::basic_ostream<char>
      &(*)(std::__1::basic_ostream<char> &)' for 1st argument
    basic_ostream& operator<<(basic_ostream& (*__pf)(basic_ostream&))
                   ^
/Library/Developer/CommandLineTools/usr/include/c++/v1/ostream:198:20: note: candidate function not viable: no known conversion from 'Fraction' to
      'basic_ios<std::__1::basic_ostream<char, std::__1::char_traits<char> >::char_type, std::__1::basic_ostream<char, std::__1::char_traits<char> >::traits_type>
      &(*)(basic_ios<std::__1::basic_ostream<char, std::__1::char_traits<char> >::char_type, std::__1::basic_ostream<char, std::__1::char_traits<char> >::traits_type> &)' (aka
      'basic_ios<char, std::__1::char_traits<char> > &(*)(basic_ios<char, std::__1::char_traits<char> > &)') for 1st argument
    basic_ostream& operator<<(basic_ios<char_type, traits_type>&
                   ^
/Library/Developer/CommandLineTools/usr/include/c++/v1/ostream:203:20: note: candidate function not viable: no known conversion from 'Fraction' to
      'std::__1::ios_base &(*)(std::__1::ios_base &)' for 1st argument
    basic_ostream& operator<<(ios_base& (*__pf)(ios_base&))
                   ^
/Library/Developer/CommandLineTools/usr/include/c++/v1/ostream:206:20: note: candidate function not viable: no known conversion from 'Fraction' to 'bool' for 1st argument
    basic_ostream& operator<<(bool __n);
                   ^
/Library/Developer/CommandLineTools/usr/include/c++/v1/ostream:207:20: note: candidate function not viable: no known conversion from 'Fraction' to 'short' for 1st argument
    basic_ostream& operator<<(short __n);
                   ^
/Library/Developer/CommandLineTools/usr/include/c++/v1/ostream:208:20: note: candidate function not viable: no known conversion from 'Fraction' to 'unsigned short' for 1st
      argument
    basic_ostream& operator<<(unsigned short __n);
                   ^
```

<< overloading

- We use **<<** to output something to an **ostream&:**

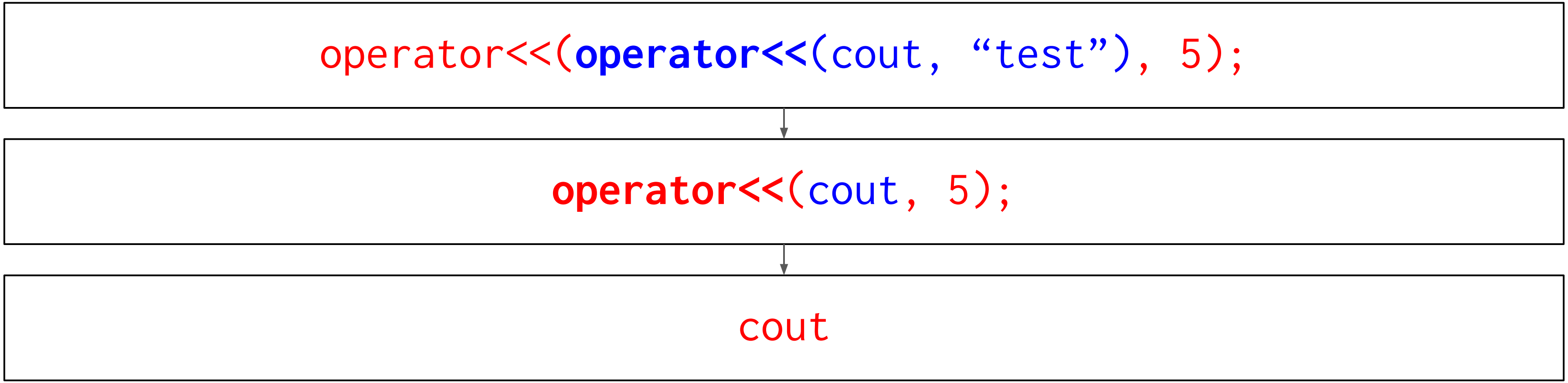
```
std::ostream& operator<<(std::ostream& out, const Time& time) {
    out << time.hours << “:” << time.minutes << “:”      // 1) print data to ostream
        << time.seconds;
    return out;                                           // 2) return original ostream
}
// in Time.h -- friend declaration allows access to private attrs
public:
     friend std::ostream& operator<<(std::ostream& out, const Time& time);

// now we can do this!
cout << t << endl;  // 5:22:31
```

This is how the magic std::cout mixing types works!

```
std::ostream& operator<<(std::ostream& out, const std::string& s);
std::ostream& operator<<(std::ostream& out, const int& i);
```

```
cout << “test” << 5;          // (cout << “test”) << 5;
```



```
operator<<(operator<<(cout, “test”), 5);
```

```
operator<<(cout, 5);
```

```
cout
```

Live Code Demo:

Fraction.cpp

Don’t overuse operator overloading

…it can be confusing

🚫 Confusing

```
MyString a("paren");
MyString b("quokka");

MyString c = a * b;  // what does this mean??
```

✔️ Clear

```
MyString a("paren");
MyString b("quokka");

MyString c = a.charsInCommon(b);  // ahh, much better
```

Rules of Operator Overloading

1. Should be **obvious** when you see it
2. Should be **reasonably similar** to corresponding arithmetic operations
    - Don’t define **+** to mean set subtraction!
3. When the meaning isn’t obvious, give it a normal name instead.

Demo: Vector.cpp

🤔 Questions? 🤔