Operator Overloading

CS 106L, Fall ‘21

How can we repurpose common operators to write descriptive and functional code?

Today's agenda

- Recap: Objects and Classes
- Operators
  - Function Overloading
  - Operator Overloading
- (that's it ! shorter lecture today c:)

Objects

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

Time

Here's an example of a class!

```
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, minutes, hours;
    // and other member (instance) variables
}
```

Time

Let's implement a simple operation, to check if 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, compare minutes
  if (a.getMinutes() < b.getMinutes()) return true;
  if (b.getMinutes() < a.getMinutes()) return false;
  // otherwise compare seconds...
}
```

Question: Why are the arguments `const`?

Usage of before()

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

Usage of before()

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

- This is somewhat hard to read.
- It's unclear whether we're checking if a is if before b, or if b is before a!

Usage of before()

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

- This is somewhat hard to read.
- It's unclear whether we're checking if a is if 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

Redefining what operators mean!

Function Overloading

Allow for calling the same function with different parameters:

```
int sum(int a, int b) {
  return a + b;
}

double sum(double a, double b) {
  return a + b;
}

// usage:
cout << sum(1.5, 2.4) << endl;
cout << sum(10, 20) << endl;
```

Operator Overloading

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

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

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

->*  new  new[]  delete  delete[]
```

Operator Overloading

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

Operator Overloading

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

Functors

```
class functor {
public:
    int operator() (int arg) const { // parameters and function body
        return num + arg;
    }
private:
    int num; // capture clause

};

int num = 0;
auto lambda = [&num] (int arg) { num += arg; };
lambda(5);
```

- Remember this?
- Check out what makes functors separate from regular classes!

Operator Overloading

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

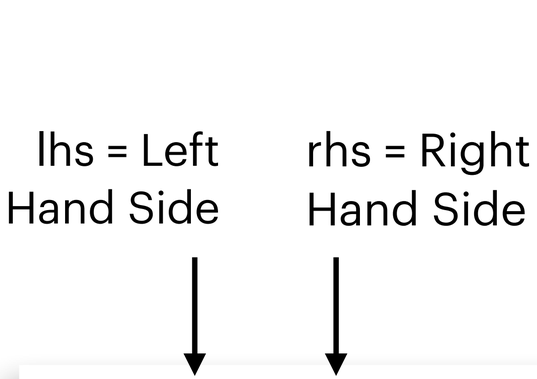
Operator Overloading

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

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

Operator Overloading

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



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

Two ways to overload operators:

1. member functions
2. non-member functions

Member Function

```
Person sathya;
sathya.enroll("Stanford");  // declared inside class Person
```

Non-Member Function

```
Person frankie;
enroll(frankie, "Stanford"); // declared globally (main.cpp?)
```

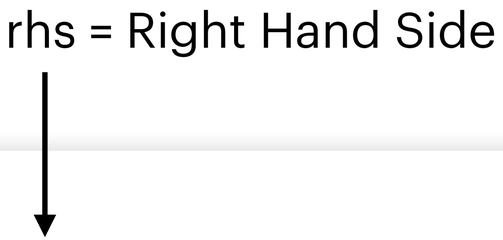
1. Member Functions

Add a function called operator _ to your class:

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

1. Member Functions

Add a function called operator _ to your class:




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

lhs (left hand side) of each operator is ***this***.

1. Member Functions

Add a function called operator __ to your class:

This...

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

becomes this!

```
Time a, b;
if (a.operator<(b)) {
  // equivalent to a < b
}
```

1. Member Functions

Add a function called operator _ to your class:

```
class Time {
  bool operator < (const Time& rhs) const;
  bool 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...
  }
}
```

🕐 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.

Let's get some practice with member function operators!

Live code demo: simple_fraction.cpp

Some issues with member function operators

Member Function

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

```
// or:
if (a < b) {
  // do something;
}
```

- Operators can only be called on the left hand side
- What if we can't control what's on the left hand side of the operation?
  - e.g. if we want to compare a double and a Fraction

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

- Instead of taking only rhs, it takes **both** the left hand side and right hand side!

2. Non-Member Functions

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

2. 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 LHS class that we don't own!

```
ExternalTime time1 (11,11,11);
MyFraction time2 (04, 04, 44);
// without non-member functions, we'd rely on ExternalTime to implement the < operator
cout << time1 < time2 << endl;
```

Without non-member functions, we'd rely on ExternalTime to implement the **`<`** operator.

2. Non-Member Functions

- You may be wondering how non-member functions can access private member variables!
- The answer: **`friend`**s!

```
class Time {
  // core member functions omitted for brevity
  public:
    friend bool operator == (const Time& lhs, const Time& rhs);
  private:
    int hours, minutes, seconds;
}

bool operator == (const Time& lhs, const Time& rhs) {
  return lhs.hours == rhs.hours && lhs.minutes == rhs.minutes && lhs.seconds == rhs.seconds;
}
```

Ever seen this?

```
Fraction a;
std::cout << a << std::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);
                   ^
```

<< Operator Overloading

We can use << to output something to an **`std::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 std::cout mixes types (and still works)!

Since these two methods are implemented in the STL,

```
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;
```

This is how std::cout mixes types (and still works)!

Since these two methods are implemented in the STL,

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

```
cout << "test" << 5; // (cout << "test") << 5;
```

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

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

```
cout;
```

This is how std::cout mixes types (and still works)!

Since these two methods are implemented in the STL,

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

```
cout << "test" << 5; // (cout << "test") << 5;
```

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

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

```
cout;
```

Let's get some practice with friend(ly) non-member function operator overloading!

Live code demo: fraction.cpp

Don't overuse operator overloading!

It can be very confusing if done wrong.

❌ Confusing

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

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

✅ Great!

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

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

Rules of Operator Overloading

1. Meaning 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.