Operator Overloading

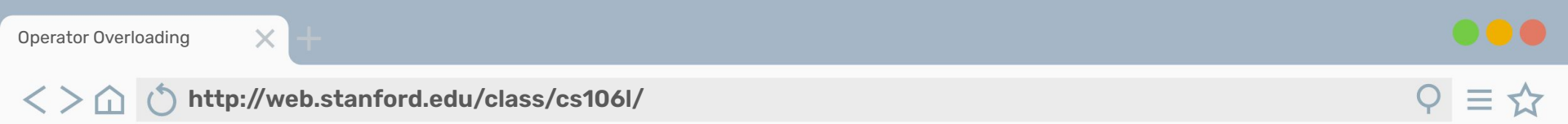
运算符重载

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

CS106L - Fall 22

Attendance!

https://bit.ly/3NsP5MH



Complete the midquarter survey!

Due next Tuesday! (11/8)

CONTENTS

http://web.stanford.edu/class/cs106l/

CONTENTS

CONTENTS

Objects and Classes

- Objects are instances of classes
- Objects encapsulate data related to a single entity
    - Define complex behavior to work with or process that data:

      `Student.printEnrollmentRecord()`

Objects and Classes

- Objects store private state through instance variables
  - `Student::name`
- Expose private state to other through public instance methods
  - `Student::getName()`
- Allow us to expose state in a way we can control

We almost have everything we need!

Classes let you define new objects with new behavior!

- We know how to parametrize classes and functions using templates!

We almost have everything we need!

Classes let you define new objects with new behavior!

- We know how to parametrize classes and functions using templates!
- But...

We almost have everything we need!

Classes let you define new objects with new behavior!

- We know how to parametrize classes and functions using templates!
- But...
- Remember maps and sets?

Unordered maps/sets

Both maps and sets in the STL have an unordered version!

- **Ordered** maps/sets require a **comparison operator** to be defined.
- **Unordered** maps/sets require a **hash function** to be defined.

Simple types are already natively supported; anything else will need to be defined yourself.

Unordered maps/sets are usually faster than ordered ones!

We almost have everything we need!

Classes let you define new objects with new behavior!

- We know how to parametrize classes and functions using templates!
- But...
- Remember maps and sets?
- And structs in streams?

A stream you've used: cout

```
std::cout << 5 << std::endl; // prints 5
// use a stream to print any primitive type!
std::cout << "Sarah" << std::endl;
// Mix types!
std::cout << "Sarah is " << 21 << std::endl;
// structs?
Student s = {"Sarah", "CA", 21};
std::cout << s << std::endl;
```

We almost have everything we need!

Classes let you define new objects with new behavior!

- We know how to parametrize classes and functions using templates!
- But...
- Remember maps and sets?
- And structs in streams?
- And functors?

Aside: What the Functor?

A **functor** is any class that provides an implementation of operator().

- They can create **closures** of "customized" functions!
- Lambdas are just a reskin of 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);
```

Closure: a single instantiation of a functor object

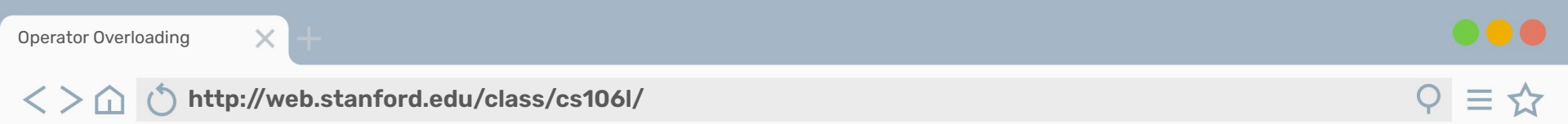
We almost have everything we need!

Classes let you define new objects with new behavior!

- We know how to parametrize classes and functions using templates!
- But...
- Remember maps and sets?
- And structs in streams?
- And functors?

We're missing something important!

CONTENTS



Let's talk about it!

How do operators work with classes?

Let's talk about it!

How do operators work with classes?

- Just like declaring functions in a class, we can declare operator functionality!

Let's talk about it!

How do operators work with classes?

- Just like declaring functions in a class, we can declare operator functionality!
- When we use that operator with our new object, it performs a custom function or operation!

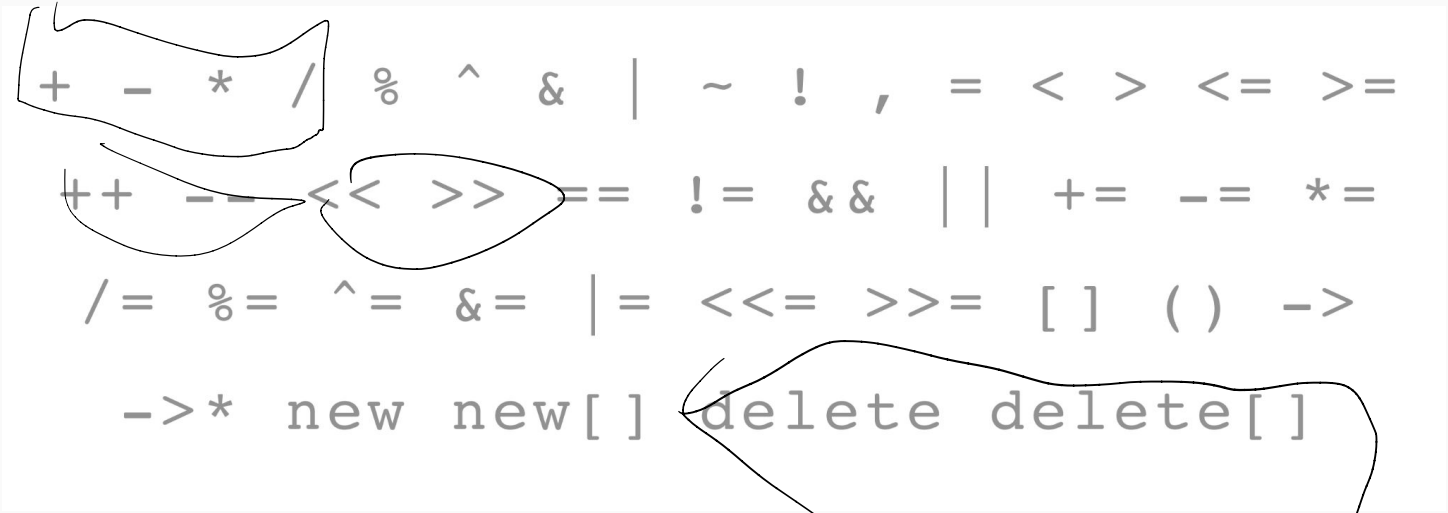
Let's talk about it!

How do operators work with classes?

- Just like declaring functions in a class, we can declare operator functionality!
- When we use that operator with our new object, it performs a custom function or operation!
- Just like in function overloading, if we give it the same name, it will override the operator's behavior!

What operators can we overload?

Most of them, actually!



What can't be overloaded?

- Scope Resolution
- Ternary
- Member Access
- Pointer-to-member access
- Object size, type, and casting

```
::    ?    .    .*    sizeof()
    typeid()    cast()
```

What can't be overloaded?

- Scope Resolution
- Ternary
- Member Access
- Pointer-to-member access
- Object size, type, and casting

```
::    ?    .    .*    sizeof()
      typeid()    cast()
```

What can't be overloaded?

- Scope Resolution
- Ternary
- Member Access
- Pointer-to-member access
- Object size, type, and casting

```
::    ?    .    .*    sizeof()
    typeid()    cast()
```

What can't be overloaded?

- Scope Resolution
- Ternary
- Member Access
- Pointer-to-member access
- Object size, type, and casting

```
::    ?    .    .*    sizeof()
    typeid()    cast()
```

these are not functions

We can go from this…

//student.h

```
class Student {
    public:
    std::string getName();
    void setName(string name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
 };
```

...to this!

//student.h

```
class Student {
    public:
    std::string getName() const;
    void setName(string name);
    int getAge() const;
    void setAge(int age);
    bool operator < (const Student& rhs) const;

    private:
    std::string name;
    std::string state;
    int age;
};
```

In the .cpp:

//student.cpp

```
#include student.h
std::string Student::getName(){
//implementation here!
}
 /* ... */

bool operator< (const Student& rhs) const {
    return age < rhs.age;
}
```

In the .cpp:

//student.cpp

```
#include student.h
std::string Student::getName(){
//implementation here!
}
 /* ... */

bool operator< (const Student& rhs) const {
    return age < rhs.age;
}
```

We're in a member function, so age refers to this->age by default!

Operator Overloading

http://web.stanford.edu/class/cs106l/

Let's take a look!

simple_fraction!

We can overload operators in two ways:

Member functions

- Declare your overloaded operator within the scope of your class!
- Allows you to use member variables of this->

We can overload operators in two ways:

Member functions

- Declare your overloaded operator within the scope of your class!
- Allows you to use member variables of this->

What if we don't know what will be on the left-hand side?

We can overload operators in two ways:

Member functions

- Declare your overloaded operator within the scope of your class!
- Allows you to use member variables of this->

Non-member functions

- Declare the overloaded operator outside of any classes (main.cpp?)
- Define both left and right hand objects as parameters

What if we don't know what will be on the left-hand side?

Non-member overloading

Non-member overloading is preferred by the STL!

Non-member overloading

Non-member overloading is preferred by the STL!

- It allows the LHS to be a non-class type (ex. comparing **double** to a **Fraction**)

Non-member overloading

Non-member overloading is preferred by the STL!

- It allows the LHS to be a non-class type (ex. comparing **double** to a **Fraction**)
- Allows us to overload operators with classes we don't own! (ex. **vector** to a **StudentList)**

Non-member overloading

Non-member overloading is preferred by the STL!

- It allows the LHS to be a non-class type (ex. comparing **double** to a **Fraction**)
- Allows us to overload operators with classes we don't own! (ex. **vector** to a **StudentList)**

```
bool operator< (const Student& lhs, const Student& rhs);
```

Non-member overloading

Non-member overloading is preferred by the STL!

- It allows the LHS to be a non-class type (ex. comparing **double** to a **Fraction**)
- Allows us to overload operators with classes we don't own! (ex. **vector** to a **StudentList)**

We now have to declare what is on the lefthand side of the operator!

```
bool operator< (const Student& lhs, const Student&
rhs);
```

What about member variables?

With member function overloading, we have access to this->

and its private variables.

What about member variables?

With member function overloading, we have access to this-> and its private variables.

Can we still access these with non-member operator overloading?

What about member variables?

With member function overloading, we have access to this->

and its private variables.

Can we still access these with non-member operator

overloading?

Everything is better with `friend`s!

The `friend` keyword allows non-member functions or classes to access private information in another class!

Everything is better with `friend`s!

The `friend` keyword allows non-member functions or classes to access private information in another class!

- To use, declare the name of the function or class as a friend within the target class's header!

Everything is better with `friend`s!

The `friend` keyword allows non-member functions or classes to access private information in another class!

- To use, declare the name of the function or class as a friend within the target class's header!
- If it's a class, you must say `friend class [name];`

//student.h

```
class Student {
    public:
    /* ... */
    friend bool operator < (const Student& lhs, const Student& rhs)
    const;

    private:
    /* ... */
};

bool operator < (const Student& lhs, const Student& rhs) {
    return lhs.age < rhs.age;
}
```

//student.h

```
class Student {
    public:
    /* ... */
    friend bool operator < (const Student& lhs, const Student& rhs)
    const;

    private:
    /* ... */
};

bool operator < (const Student& lhs, const Student& rhs) {
    return lhs.age < rhs.age;
}
```

Seen this before?

This happens when a custom class hasn't defined the stream operator!

```
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);
                   ^
```

We can do something like this!

Operator overloading is how the STL lets cout mix types!

```
std::ostream& operator << (std::ostream& out, const Time& time) {
    out << time.hours << ":" << time.minutes << ":" << time.seconds;
    return out;
}
```

Be careful with non-member overloading!

Certain operators, like `new` and `delete`, don't require a specific type.

- Overloading this outside of a class is called **global overloading** and will affect everything!

```
void* operator new(size_t size);
```

Overloading Strategies

As with everything, there's a time and a place for operator overloading!

- **Don't go overboard**; it can be confusing if overused.

Compare:

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

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

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

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

Rules and Philosophy

- Meaning should be **obvious** when you see it

Rules and Philosophy

- Meaning should be **obvious** when you see it
- Functionality should be **reasonably similar** to corresponding arithmetic operations
    - Don't define + to mean set subtraction!

Rules and Philosophy

- Meaning should be **obvious** when you see it
- Functionality should be **reasonably similar** to corresponding arithmetic operations
    - Don't define + to mean set subtraction!
- When the meaning isn't obvious, give it a normal name instead.

http://web.stanford.edu/class/cs106l/

Thanks!

Next up: participating in the democratic process of a free nation!

(... and then special member functions)