Lecture 12: Operator Overloading

Stanford CS106L, Spring 2026

Rachel Fernandez & Preston Seay

Attendance

Today's Agenda

1. Recap
2. Operator Overloading

Today's Agenda

1. **Recap**
2. Operator Overloading

Functors

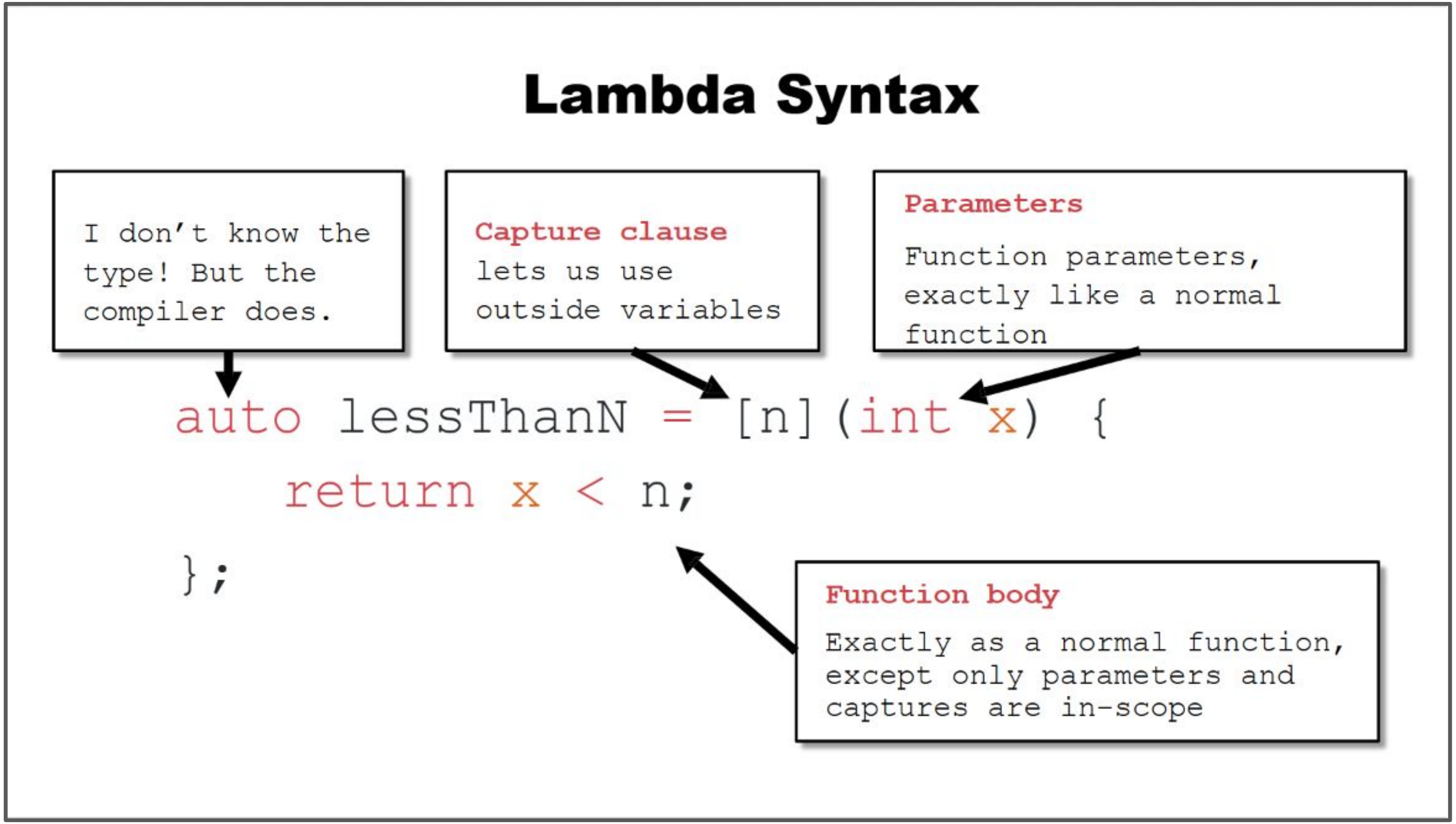
Containers
How do we store groups of things?

Iterators
How do we traverse containers?

Functors
How can we represent functions as objects?

Algorithms
How do we transform and modify containers in a generic way?

Functors


Lambda Syntax
I don’t know the type! But the compiler does.
Capture clause
lets us use outside variables
Parameters
Function parameters, exactly like a normal function
auto lessThanN = [n](int x) {
return x < n;
};
Function body
Exactly as a normal function, except only parameters and captures are in-scope

Algorithms

Containers
How do we store groups of things?

Iterators
How do we traverse containers?

Functors
How can we represent functions as objects?

Algorithms
How do we transform and modify containers in a generic way?

Ranges and Views

We can chain views together use operator |

```
std::vector<char> letters = {'a','b','c','d','e'};
std::vector<char> upperVowel = letters
  |  std::ranges::views::filter(isVowel)
  |  std::ranges::views::transform(toupper)
  |  std::ranges::to<std::vector<char>>();

// upperVowel = { 'A', 'E' }
```

It’s week 6!

C++ reference

C++11, C++14, C++17, C++20, C++23, C++26 | Compiler support C++11, C++14, C++17, C++20, C++23, C++26

Language
- Keywords − Preprocessor
- ASCII chart
- Basic concepts
  - Comments
  - Names (lookup)
  - Types (fundamental types)
  - The main function
- Expressions
  - Value categories
  - Evaluation order
  - Operators (precedence)
  - Conversions − Literals
- Statements
  - if − switch
  - for − range-for (C++11)
  - while − do-while
- Declarations − Initialization
- Functions − Overloading
- Classes (unions)
- Templates − Exceptions
- Freestanding implementations

Standard library (headers)

Named requirements

Feature test macros (C++20)
- Language − Standard library

Language support library
- Program utilities
  - Signals − Non-local jumps
- Basic memory management
- Variadic functions
- source_location (C++20)
- Coroutine support (C++20)
- Comparison utilities (C++20)
- Type support − type_info
- numeric_limits − exception
- initializer_list (C++11)

Concepts library (C++20)

Diagnostics library
- Assertions − System error (C++11)
- Exception types − Error numbers
- basic_stacktrace (C++23)
- Debugging support (C++26)

Memory management library
- Allocators − Smart pointers
- Memory resources (C++17)

Metaprogramming library (C++11)
- Type traits − ratio
- integer_sequence (C++14)

General utilities library
- Function objects − hash (C++11)
- Swap − Type operations (C++11)
- Integer comparison (C++20)
- pair − tuple (C++11)
- optional (C++17)
- expected (C++23)
- variant (C++17) − any (C++17)
- bitset − Bit manipulation (C++20)

Containers library
- vector − deque − array (C++11)
- list − forward_list (C++11)
- map − multimap − set − multiset
- unordered_map (C++11)
- unordered_multimap (C++11)
- unordered_set (C++11)
- unordered_multiset (C++11)
- Container adaptors
- span (C++20) − mdspan (C++23)

Iterators library

Ranges library (C++20)
- Range factories − Range adaptors
- generator (C++23)

Algorithms library
- Numeric algorithms
- Execution policies (C++17)
- Constrained algorithms (C++20)

Strings library
- basic_string − char_traits
- basic_string_view (C++17)
- Null-terminated strings:
  - byte − multibyte − wide

Text processing library
- Primitive numeric conversions (C++17)
- Formatting (C++20)
- locale − Character classification
- text_encoding (C++26)
- Regular expressions (C++11)
  - basic_regex − Algorithms
  - Default regular expression grammar

Numerics library
- Common math functions
- Mathematical special functions (C++17)
- Mathematical constants (C++20)
- Basic linear algebra algorithms (C++26)
- Pseudo-random number generation
- Floating-point environment (C++11)
- complex − valarray

Date and time library
- Calendar (C++20) − Time zone (C++20)

Input/output library
- Print functions (C++23)
- Stream-based I/O − I/O manipulators
- basic_istream − basic_ostream
- Synchronized output (C++20)
- File systems (C++17)

Concurrency support library (C++11)
- thread − jthread (C++20)
- atomic − atomic_flag
- atomic_ref (C++20) − memory_order
- Mutual exclusion − Semaphores (C++20)
- Condition variables − Futures
- latch (C++20) − barrier (C++20)
- Safe Reclamation (C++26)

Execution support library (C++26)

Technical specifications
- **Standard library extensions** (library fundamentals TS)
  - resource_adaptor − invocation_type
- **Standard library extensions v2** (library fundamentals TS v2)
  - propagate_const − ostream_joiner − randint
  - observer_ptr − Detection idiom
- **Standard library extensions v3** (library fundamentals TS v3)
  - scope_exit − scope_fail − scope_success − unique_resource

Parallelism library extensions v2 (parallelism TS v2)
- simd

Concurrency library extensions (concurrency TS)

Transactional Memory (TM TS)

Reflection (reflection TS)

External Links − Non-ANSI/ISO Libraries − Index − std Symbol Index

We've made it really far

Week	Tuesday	Thursday
1	March 31 1. Welcome! 📖 Slides 📃 Policies	April 2 2. Types & Structs 📖 Slides 💻 Code
2	April 7 3. Initialization & References 📖 Slides	April 9 4. Streams 📖 Slides 💻 Code A1: SimpleEnroll
3	April 14 5. Containers 📖 Slides 💻 Code	April 16 6. Iterators & Pointers 📖 Slides 💻 Code A2: Marriage Pact
4	April 21 7. Classes 📖 Slides 💻 Code	April 23 8. Optional: Inheritance Practice 📖 Slides 💻 Code A3: Make a Class!
5	April 28 9. Class Templates & Const Correctness 📖 Slides 💻 Code	April 30 10. Function Templates 📖 Slides 💻 Code A4: Ispell
6	May 5 11. Functions & Lambdas 📖 Slides	May 7 12. Operator Overloading

What questions do we have?

Today's Agenda

1. Recap
2. **Operator Overloading**

So what have we seen so far

At this point:

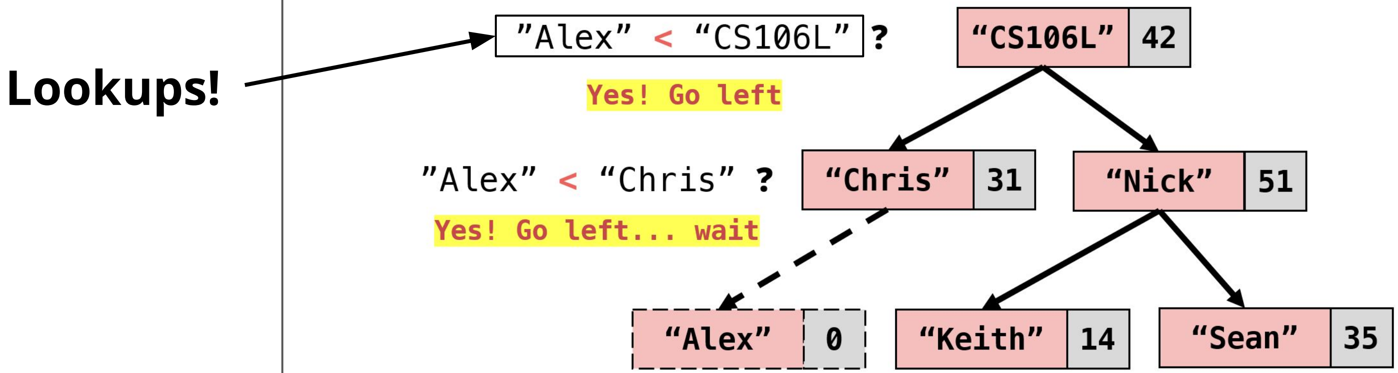
1. You know how to create classes!
2. You know to to create *templated* classes!
3. But.....
4. Remember **`maps`** and **`sets`**?

In particular recall that a **`std::map<K ,V>`** requires **`K`** to have an **`operator<`**

Why this requirement?

In particular recall that a **std::map<K ,V>** requires **K** to have an **operator<**

What is map[“Alex”]?



Motivation

Why should we use operators at all?

“Operators allow you to convey meaning about types that functions don’t”

From this this phenomenal cppcon video

Motivation

Why should we use operators at all?

“Operators allow you to convey meaning about types that functions don’t”

```
class Money {
public:
    int cents;
    Money(int c) : cents(c) {}
};
```

Motivation

Why should we use operators at all?

“Operators allow you to convey meaning about types that functions don’t”

```
class Money {
public:
    int cents;
    Money(int c) : cents(c) {}
};
```

```
Money add(const Money& a, const Money& b) {
    return Money(a.cents + b.cents);
}

Money total = add(Money(100), Money(50));  // 100 + 50 = 150
```

Feels like a random function call.. Not really addition

Motivation

Why should we use operators at all?

“Operators allow you to convey meaning about types that functions don’t”

```cpp
class Money {
public:
    int cents;
    Money(int c) : cents(c) {}
};
```

```cpp
Money add(const Money& a, const Money& b) {
    return Money(a.cents + b.cents);
}

Money total = add(Money(100), Money(50));  // 100 + 50 = 150
```

```cpp
Money operator+(const Money& a, const Money& b) {
    return Money(a.cents + b.cents);
}

Money total = Money(100) + Money(50);
```

Now I understand! Money has a numeric-like behavior because we understand the + symbol means you can add them!

From this this phenomenal cppcon video

Hey Bjarne, I want the min of 2 ???

```
template <typename T>
T min(const T& a, const T& b) {
  return a < b ? a : b;
}

// For which T will the following compile successfully?
T a = /* an instance of T */;
T b = /* an instance of T */;
min<T>(a, b);
```

What **must be true** of a type T for us to be able to use min?

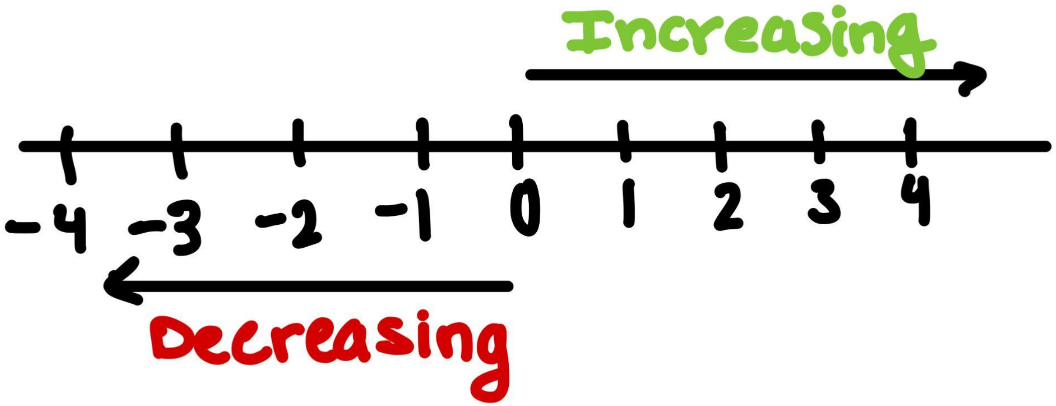
Hey Bjarne, I want the min of 2 ???

What **must be true** of a type T for us to be able to use min?

1. T should have an ordering relationship that makes sense.

2. T should represent something **comparable** where a “minimum” can be logically determined

Hey Bjarne, I want the min of 2 int



1. T should have an ordering relationship that makes sense.

2. T should represent something **comparable** where a “minimum” can be logically determined

Hey Bjarne, I want the min of 2 StanfordIDs

```
StanfordID rachel;
StanfordID preston;

auto minStanfordID = min<StanfordID>(preston, rachel);
```

Hey Bjarne, I want the min of 2 StanfordIDs

```
StanfordID rachel;
StanfordID preston;

auto minStanfordID = min<StanfordID>(rachel, preston);


StanfordID min(const StanfordID& a, const StanfordID& b)
{
    return a < b ? a : b;
}
```

Compiler: "Hey, I don't know what to do here!"

Hello Operator Overloading

So how do operators work with classes?

- Just like we declare functions in a class, we can declare an operator’s 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 are operators?

Operators are symbols that perform operations on values, objects, or types and produce a new value or effect.

Values

```
3 + 4
```

Objects

```
Point a;
Point b;
a + b
```

Types

```
sizeof(int)
new int(5)
```

What operators can we overload?

It turns out, most of them!

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

What operators can't be overloaded?

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

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

What operators can't be overloaded?

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

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

What operators can't be overloaded?

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

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

What operators can't be overloaded?

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

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

Operator Overloading Syntax

```
return_type operator<symbol>(parameter_list);
```

Hey Bjarne, I want the min of 2 StanfordIDs

.h file

```
class StanfordID {
private:
std::string name;
std::string sunet;
int idNumber;

public:
    // constructor for our StanfordID
    StanfordID(std::string name, std::string sunet, int idNumber);
    std::string getIdNumber();
    .
    .
    bool operator < (const StanfordID& other) const;
}
```

Hey Bjarne, I want the min of 2 StanfordIDs

.cpp file

```
#include StanfordID.h

std::string StanfordID::getIdNumber() {
    return idNumber;
}


bool StanfordID::operator < (const StanfordID& rhs) const {
   ?
}
```

Think about it with a partner!

Say that you want to compare `StanfordID` objects by their `idNumber` member variable, how could you implement this?

```
bool operator< (const StudentID& rhs) const {
  // TODO: compare StudentIDs by their idNumbers
}
```

Hey Bjarne, I want the min of 2 StanfordIDs

.cpp file

```
#include StanfordID.h

int StanfordID::getIdNumber() {
    return idNumber;
}


bool StanfordID::operator<(const StanfordID& other) const {
    return idNumber < other.getIdNumber();
}
```

Hey Bjarne, I want the min of 2 StanfordIDs

.cpp file

```
#include StanfordID.h

int StanfordID::getIdNumber() {
    return idNumber;
}


bool StanfordID::operator<(const StanfordID& other) const {
    return idNumber < other.idNumber;
}
```

What questions do we have?

Practice

https://106b.vercel.app/cs106l-operator-overloading

Non-member overloading

There are two ways to overload:

1. **Member overloading**
    a. Declares the overloaded operator within the scope of your class

Non-member overloading

There are two ways to overload:

1. Member overloading
   a. Declares the overloaded operator within the scope of your class

This is what we've seen!

Non-member overloading

There are two ways to overload:

1. **Member overloading**
    a. Declares the overloaded operator within the scope of your class

.h file

```
class StanfordID {
private:
std::string name;
std::string sunet;
int idNumber;

public:
    // constructor for our StanfordID
    StanfordID(std::string name, std::string sunet, int idNumber);
    std::string getIdNumber();
    .
    .
    bool operator < (const StanfordID& other) const;
}
```

Non-member overloading

There are two ways to overload:

1. **Member overloading**
    a. Declares the overloaded operator within the scope of your class

2. **Non-member overloading**
    a. Declare the overloaded operator outside of class definitions
    b. Define both the left and right hand objects as parameters

Non-member overloading

Non-member Operator Overloading

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

Member Operator Overloading

```
bool StanfordID::operator < (const StanfordID& rhs) const {...}
```

Why non-member overloading?

```cpp
class StanfordID {
private:
    std::string sunet;
public:
    StanfordID(std::string s) : sunet(s) {}

    bool operator<(const std::string& other) const {
        return sunet < other;
    }
};
```

```cpp
StanfordID rachel("rfer");
std::string name = "zzhang";

if (rachel < name) {
    std::cout << "Rachel comes before name\n";
}
```

Why non-member overloading?

```
class StanfordID {
private:
    std::string sunet;
public:
    StanfordID(std::string s) : sunet(s) {}

    bool operator<(const std::string& other) const {
        return sunet < other;
    }
};
```

```
StanfordID rachel("rfer");
std::string name = "zzhang";

if (name < rachel) {
    std::cout << "Name comes before Rachel\n";
}
```

Non-member overloading

```
StanfordID rachel("rfer");
std::string name = "zzhang";

if (name < rachel) {
    std::cout << "Name comes before Rachel\n";
}
```

```
name.operator<(rachel);   // tries to call string's member function
```

Non-member overloading

This is actually preferred by the STL, and is more idiomatic C++

Why:

1. Allows for the **left-hand-side** to be a **non-class type**

```
bool operator<(int lhs, const StanfordID& rhs) {
    return lhs < rhs.getIDNumber();
}
```

Non-member overloading

This is actually preferred by the STL, and is more idiomatic C++

Why:

2. Allows us to overload operators with classes we don’t own
   a. We could define an operator to compare a `StanfordID` to other custom classes you define.

```
class StanfordID {
private:
    std::string sunet;
public:
    StanfordID(std::string s) : sunet(s) {}

    bool operator<(const std::string& other) const {
        return sunet < other;
    }
};
```

```
StanfordID rachel("rfer");
std::string name = "zzhang";

if (rachel < name) {
    std::cout << "Rachel comes before name\n";
}
```

Non-member overloading

This is actually preferred by the STL, and is more idiomatic C++

Why:

1. Allows us to overload operators with classes we don't own
    a. We could define an operator to compare a `StanfordID` to other custom classes you define.

```
class StanfordID {
private:
    std::string sunet;
public:
    StanfordID(std::string s) : sunet(s) {}

    bool operator<(const std::string& other) const {
        return sunet < other;
    }
};
```

```
StanfordID rachel("rfer");
std::string name = "zzhang";

if (name < rachel) {
    std::cout << "Name comes before Rachel\n";
}
```

Non-member overloading

```
class StanfordID {
private:
    std::string sunet;
public:
    StanfordID(std::string s) : sunet(s) {}
    std::string getSunet() const { return sunet; }
};

// Non-member operator
bool operator<(const StanfordID& lhs, const std::string& rhs) {
    return lhs.getSunet() < rhs;
}

// And if you want symmetry:
bool operator<(const std::string& lhs, const StanfordID& rhs) {
    return lhs < rhs.getSunet();
}
```

It’s better to use non-member overloading so we can do comparison in both directions and with classes we don’t own!

Non-member overloading

Non-member Operator Overloading

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

Note both the left and right hand side of the operator are passed in in non-member operator overloading!

```
bool StanfordID:                                              ..}
```

What about the member variables?

Non-member Operator Overloading

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

With member operator overloading we have access to **this->** and the **variables of the class.**

.cpp file

```
#include StanfordID.h

int StanfordID::getIdNumber() {
    return idNumber;
}


bool StanfordID::operator<(const StanfordID& other) const {
    return idNumber < other.idNumber;
}
```

What about the member variables?

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

What about the member variables?

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

What about the member variables?

Non-member Operator Overloading

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

Member Operator Overloading

```
bool StanfordID::operator < (const StanfordID& rhs) const {...}
```

It is also undefined behavior to have both of these because the < operator is acting on two `StanfordID`s

Remember ambiguity badddddd

What questions do we have?

Hello `friend`!

Non-member Operator Overloading

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

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

Hello `friend`!

Non-member Operator Overloading

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

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

How do you use `friend`?
In the header of the target class you declare the operator overload function as a `friend`

Hey Bjarne, I want the min of 2 StanfordIDs

.h file

```
class StanfordID {
private:
std::string name;
std::string sunet;
int idNumber;

public:
    // constructor for our StudentID
    StanfordID(std::string name, std::string sunet, int idNumber);
    .
    .
    .
    friend bool operator < (const StanfordID& lhs, const StanfordID& rhs);
}
```

Hey Bjarne, I want the min of 2 StanfordIDs

.cpp file

```
#include StanfordID.h

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

Note: this also works!

.cpp file

```
#include StanfordID.h

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

In this case the friend keyword is not required since we're not using a private member function or variable

What questions do we have?

So why is this even meaningful?

```
StanfordID rachel;
StanfordID preston;

auto minStanfordID = min<StanfordID>(rachel, preston);


StanfordID min(const StanfordID& a, const StanfordID& b)
{
    return a < b ? a : b;
}
```

Compiler: "Hey, now I know what to do here! 😀"

So why is this even meaningful?

- There are many operators that you can define in C++ like we saw

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

So why is this even meaningful?

- There are many operators that you can define in C++ like we saw
- There’s a lot of functionality we can unlock with operators

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

More importantly

“Operators allow you to convey meaning about types that functions don’t”

Rules and Philosophies

- Because operators are intended to convey meaning about a type, the meaning should be **obvious**

- The operators that we can define are oftentimes arithmetic operators. The functionality should be **reasonably similar** to their corresponding operations
  - You don't want to define `operator+` to be set subtraction

- If the meaning is not obvious, then maybe define a function for this

This is known as the Principle of Least Astonishment (PoLA)

In general

- There are some good practices like the **rule of contrariety**
- For example when you define the `operator==` use the rule of contrariety to define `operator!=`

```
bool StanfordID::operator==(const StanfordID& other) const {
    return (name == other.name) && (sunet == other.sunet) &&
    (idNumber == other.idNumber);
}

bool StanfordID::operator!=(const StanfordID& other) const {
    return !(*this == other);
}
```

<<

- However there's a lot of flexibility in implementing operators
- For example << stream insertion operator

```
std::ostream& operator << (std::ostream& out, const StanfordID& sid) {
    out << sid.name << " " << sid.sunet << " " << sid.idNumber;
    return out;
}

std::ostream& operator << (std::ostream& out, const StanfordID& sid) {
    out << "Name: " << sid.name << " sunet: " <<                          "
    << sid.idNumber;
    return out;
}
```

The way you use this operator may influence how you implement it

Practice

Pizza Order Class

Variables: customer, topping, num of slices

Getter functions:

getCustomer()
getTopping()
getSlices()

Operator Overloading Functions:

+= → add more slices to someone's order

== → check if the number of slices, customer, and toppings are the exact same

< → check if one pizza order has less slices than the other

> → check if one pizza order has more slices than the other

<< → print CustomerName: #ofSlices, Topping

Practice

106b.vercel.app/cs106l-operator-overloading-2

Final thoughts

1. Operator overloading unlocks a new layer of functionality and meaning within objects that we define

2. Operators should *make sense*, the entire point is that convey some meaning that functions don’t about the type itself.

3. You should overload when you need to, for example if you’re not using a stream with your type, then don’t overload << or >>.