

Day 11

Operator Overloading

Preston Seay & Emily Ai

Lambdas & Functors Recap

Q: What does the compiler turn a lambda into?

```
1 auto isStrong = [](int power) { return power > 80; };
```

A functor: an object that overloads operator()!

```
1 class __lambda {  
2 public:  
3     bool operator()(int power) const { return power > 80; }  
4 };
```

Wait... we can overload operators?! 🤔

Warm-up - Pick Your Superpower



<https://www.106l.com/live/operator-overloading/warm-up?key=JREj1WV9MG8sfRQpl9GxURQeOcNVNEYq--7S6j8N6fw> ↗

Announcements!

1. Midterm Project is out, due Friday!

Plan

1. **Motivation**
2. What can we overload?
3. Member vs. Non-member
4. The `friend` keyword
5. The `static` keyword

Hey Bjarne, I want the min of 2 Heroes

```
1 Hero captain{"Captain Const", 70};  
2 Hero lass{"Lambda Lass", 90};  
3  
4 Hero weaker = std::min(captain, lass);
```



```
1 template <typename T>  
2 const T& min(const T& a, const T& b) {  
3     return (b < a) ? b : a;  
4 }
```

Compiler: "What does < even mean for a Hero?!"

Same problem: `std::sort`, `std::set<Hero>`, `std::map<Hero, V>...`

Why Operators?

"Operators allow you to convey meaning about types that functions don't"

```
1 Hero team = captain.teamUpWith(lass);  
2 if (captain.isWeakerThan(lass)) { ... }
```

Feels like random
function calls...

```
1 Hero team = captain + lass;  
2 if (captain < lass) { ... }
```

Instantly readable:
heroes can be added
and compared!

(from a great CppCon talk on operator overloading)

What Operators Can We Overload?

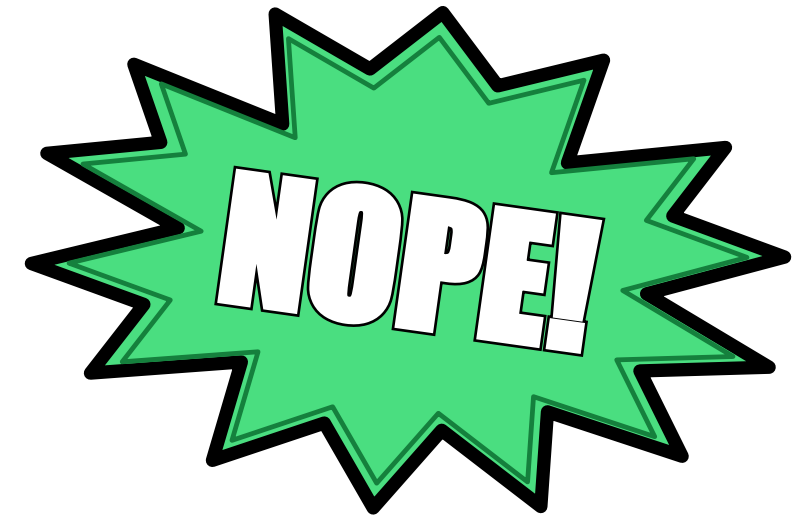
Almost all of them!

1	+	-	*	/	%	^	&		~	!
2	=	<	>	+=	-=	*=	/=	%=	^=	&=
3	=	<<	>>	>>=	<<=	==	!=	<=	>=	<=>
4	&&		++	--	,	->*	->	()	[]	
5	new	new[]	delete	delete[]						

() is how functors work!

Kryptonite: Operators You Can't Overload

1	::	scope resolution
2	? :	ternary
3	.	member access
4	.*	pointer-to-member access
5	sizeof() typeid()	casts



These work on **names and types**, not values. Changing them would break the language itself.

Operator Overloading Syntax

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

operator + the symbol

doesn't change this

```
1 bool operator<(const Hero& other) const;
```

return type

the right-hand side

Member Overloading: hero.h

```
1 class Hero {
2 private:
3     std::string name;    // secret identity!
4     int power;
5
6 public:
7     Hero(std::string name, int power);
8     std::string describe() const;
9
10    bool operator<(const Hero& other) const;
11 };
```



Declared **inside** the class, just like any other member function.

Think About It With a Partner!

Compare heroes by power. How do we implement this?

```
1 bool Hero::operator<(const Hero& other) const {  
2     ?  
3 }
```

We can read
other.power
because we are
inside Hero!

captain < lass means captain.operator<(lass)

Activity: Power Rankings

106l.com → Operator Overloading → slides 1 & 2

Plan

1. Motivation
2. What can we overload?
3. **Member vs. Non-member**
4. The `friend` keyword
5. The `static` keyword

Two Ways to Overload

Member

Inside the class. The left side is this.

```
1 bool Hero::operator<(const Hero& rhs) const {  
2     return power < rhs.power;  
3 }
```

$a < b \rightarrow a.operator<(b)$

Non-member

Outside any class. Both sides are parameters.

```
1 bool operator<(const Hero& lhs, const Hero& rhs) {  
2     return lhs.power < rhs.power;  
3 }
```

$a < b \rightarrow operator<(a, b)$

Why Non-member?

Preferred by the STL, and more idiomatic C++:

#1 The left side can be a type that isn't your class

```
1 Hero buffed = 2 * captain; // 2 is an int: no members!
```

#2 Overload with classes we don't own

```
1 std::cout << captain; // can't add members to std::ostream!
```

#3 Both sides are treated the same way

But... What About Secret Identities?

```
1 Hero operator+(const Hero& lhs, const Hero& rhs) {  
2     return Hero{lhs.name + " & " + rhs.name,  
3         lhs.power + rhs.power};  
4 }
```

Can a non-member read private members?

```
1 error: 'name' is a private member of 'Hero'
```

Also: don't write both a member **and** a non-member $\leftarrow$. The call becomes **ambiguous!**

Activity: Team Up!

106l.com → Operator Overloading → slide 3

Plan

1. Motivation
2. What can we overload?
3. Member vs. Non-member
4. **The `friend` keyword**
5. The `static` keyword

Hello, friend!

`friend` lets a non-member function (or class) access a class's **private** members.

```
1 class Hero {  
2 private:  
3     std::string name;  
4     int power;  
5  
6 public:  
7     Hero(std::string name, int power);  
8     // ...  
9     friend Hero operator+(const Hero& lhs, const Hero& rhs);  
10 };
```

Declared **inside** the class: the class grants friendship.

Still a non-member.

With **public getters**, `friend` isn't needed at all!

Live Action: Assemble the Operators!

Cast

Hero A & Hero B:

a public name sign, plus a sealed private envelope (power level)

 **operator+:**

lives *outside* the class

 **The Compiler:**

guards every envelope

Act 1: $a < b$ (member)

Hero A *is* the operator. Members may open any Hero's envelope.

Act 2: $a + b$ (non-member)

operator+ reaches for an envelope...



"ERROR: private member of Hero!"

Act 3: friend

The heroes hand **operator+** a VIP pass.
Open, add, and a new team-up Hero is born!

Activity: Friends in High Places

106l.com → Operator Overloading → slide 4

With Great Power Comes Great Responsibility

- Operators should mean what everyone **expects** them to mean
- Don't make `operator+` do set subtraction!
- If the meaning isn't obvious, write a named function instead

Principle of Least Astonishment (PoLA)

Rule of contrariety: define "not equal" in terms of `==`

```
1 bool operator!=(const Hero& lhs, const Hero& rhs) {  
2     return !(lhs == rhs);  
3 }
```

The Classic Friend: `operator<<`

```
1 // hero.h, inside class Hero
2 friend std::ostream& operator<<(std::ostream& out, const Hero& hero);
3
4 // hero.cpp
5 std::ostream& operator<<(std::ostream& out, const Hero& hero) {
6     out << hero.name << " (power " << hero.power << ")";
7     return out;
8 }
9
10 // Anywhere else (in the main function, for example)
11 std::cout << captain << " teams up with " << lass << "\n";
```

Non-member: the left side is an `std::ostream`, which we don't own.

Return the stream so `<<` can **chain**.

Plan

1. Motivation
2. What can we overload?
3. Member vs. Non-member
4. The `friend` keyword
5. The **`static`** keyword

How Many Heroes Are There?

Goal: count every Hero that has ever been created.

```
1 class Hero {  
2     // ...  
3     int heroCount = 0;  
4 };  
5  
6 Hero::Hero(/* ... */) {  
7     heroCount++;  
8 }
```

Every hero gets their **own** heroCount... so each one is just 1! 😞

We want **one** variable shared by the **whole** class.

static Member Variables

```
1 class Hero {  
2     std::string name;  
3     int power;  
4     static int heroCount; // ONE copy, shared  
5     // ...  
6 };
```

Belongs to the **class**,
not to any one object.

```
1 int Hero::heroCount = 0; // define it once  
2  
3 Hero::Hero(/* ... */) {  
4     heroCount++;  
5 }
```

Defined **once**, outside the
class (in `hero.cpp`).

C++17: **inline static**
lets you define it right in
the header!

static Member Functions

```
1 class Hero {
2 public:
3     static int getHeroCount();
4 };
5
6 int Hero::getHeroCount() { // no 'static' here!
7     return heroCount;
8 }
```

```
1 std::cout << Hero::getHeroCount(); // 0
2 Hero captain{"Captain Const", 70};
3 Hero lass{"Lambda Lass", 90};
4 std::cout << Hero::getHeroCount(); // 2
```

Called on the **class**:
Hero::getHeroCount().
No object!

No **this**! It can only
touch **static** members.

Write **static** only on
the **declaration**, not
the definition.

Think About It With a Partner!

Will this compile? Why or why not?

```
1 int Hero::getHeroCount() {  
2     return power;  
3 }
```

Whose power? A static function isn't called on any hero, so there's no **this!**

```
1 error: invalid use of member 'power'
```

Regular member functions *can* still use `static` members, though!

static Wears Many Capes

In a class

Shared by every object.

```
1 static int heroCount; // one copy for all
```

In a function

Keeps its value between calls.

```
1 int rescueCount() {  
2     static int count = 0; // initialized once  
3     return ++count;  
4 }
```

At file scope

Only visible inside this .cpp file.

```
1 static void secretLair(); // this file only
```

Final Thoughts

1. Operator overloading lets our types act like built-in types
2. Operators should convey **obvious** meaning
3. Prefer **non-member**. Add `friend` when it needs private members
4. Only overload what you need
5. **static** members belong to the **class**, not any one object