So what have we seen so far

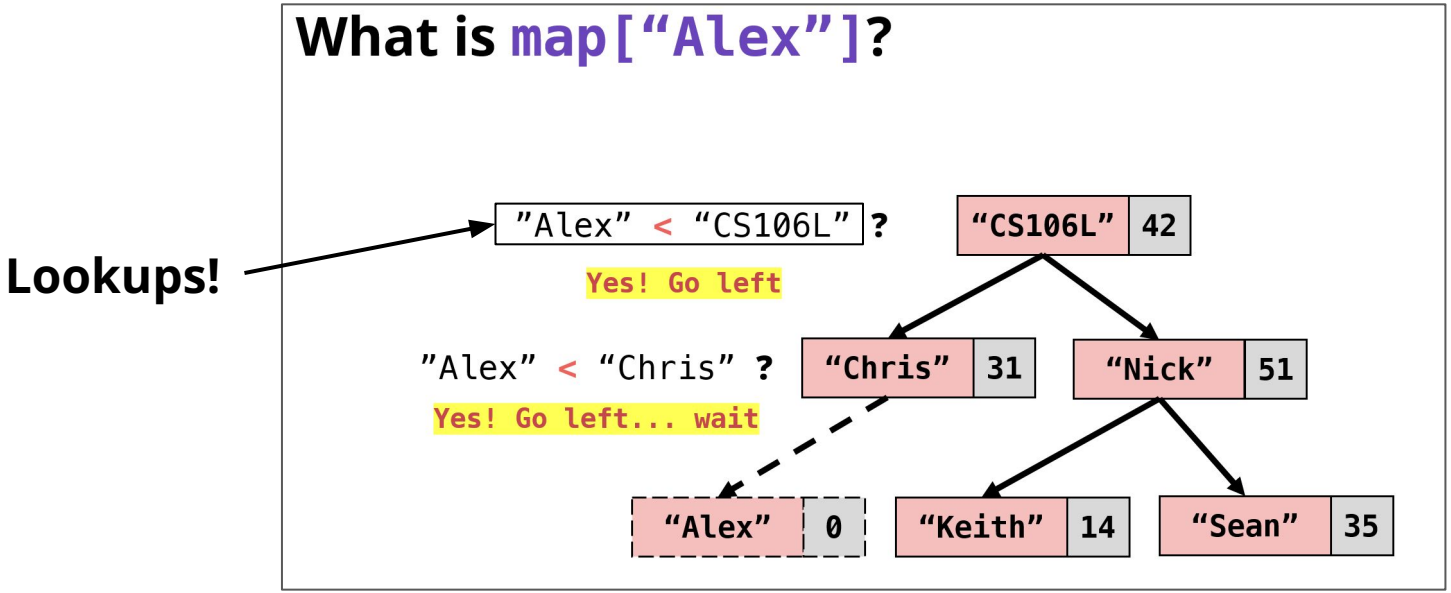
At this point:

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<`



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**?

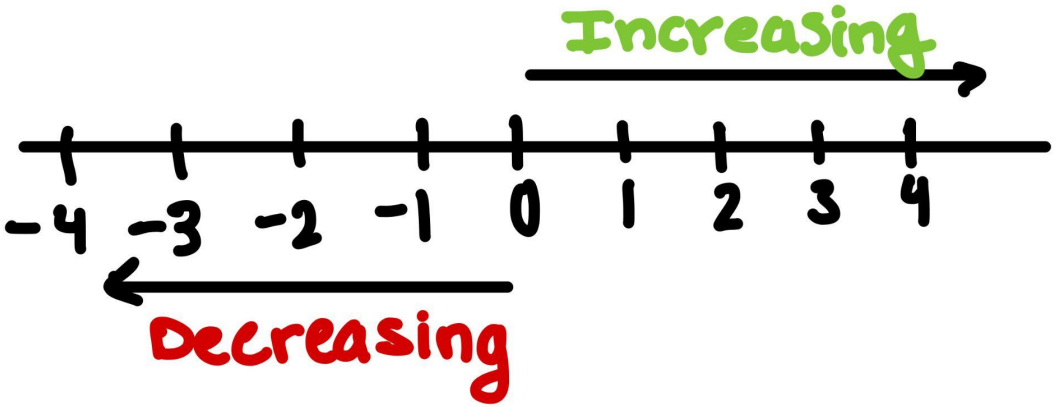
Hey Bjarne, I want the min of 2 ???

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**, ordered concept, 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**, ordered concept, where a "minimum" can be logically determined

Hey Bjarne, I want the min of 2 StanfordIDs

```
StudentID jacob;
StudentID fabio;

auto minStanfordID = min<StanfordID>(jacob, fabio);


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 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()
```

Hey Bjarne, I want the min of 2 StanfordIDs

.h file

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

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

Hey Bjarne, I want the min of 2 StanfordIDs

.cpp file

```
#include StudentID.h

std::string StudentID::getName() {
    // implementation here
}


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

Hey Bjarne, I want the min of 2 StanfordIDs

.cpp file

```cpp
#include StudentID.h

std::string StudentID::getName() {
    // implementation here
}


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

Non-member overloading

There are two ways to overload:

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

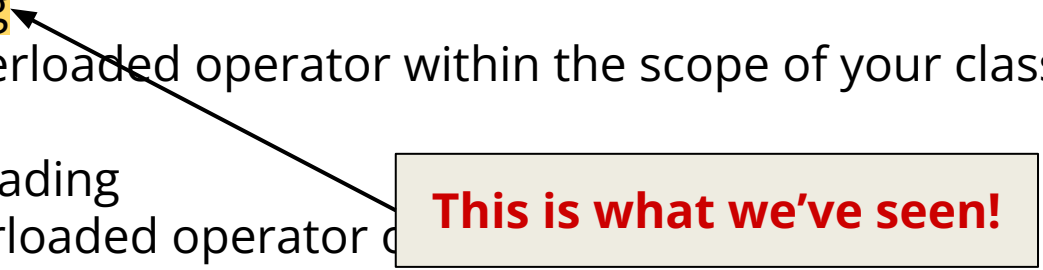
2. Non-member overloading
    a. Declare the overloaded operator outside of class definitions
    b. Define both the left and right hand objects as parameters

Non-member overloading

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 o
   b. Define both the left and right hand objects as parameters



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**

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

Non-member overloading

Non-member Operator Overloading

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

Member Operator Overloading

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

What about the member variables?

Non-member Operator Overloading

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

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

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

Hello `friend`!

Non-member Operator Overloading

```
bool operator< (const StudentID& lhs, const StudentID& 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 StudentID {
private:
std::string name;
std::string sunet;
int idNumber;

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

Hey Bjarne, I want the min of 2 StanfordIDs

.cpp file

```
#include StudentID.h

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

So why is this even meaningful?

```
StudentID jacob;
StudentID fabio;

auto minStanfordID = min<StanfordID>(jacob, fabio);


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
- 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”

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

bool StudentID::operator!=(const StudentID& 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 StudentID& sid) {
    out << sid.name << " " << sid.sunet << " " << sid.idNumber;
    return out;
}

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

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