Template and Iterator Classes

CS 106L Winter 2020 – Avery Wang and Anna Zeng

Game Plan

- operators and friends (15)
- POLA (5)
- template classes (10)
- iterator classes (20)

operator overloading

How does C++ know how to apply operators to user-defined classes?

```
vector<string> v{“Hello”, “World”};
cout << v[0];
v[1] += “!”;
```

C++ tries to call these functions.

```
vector<string> v{“Hello”, “World”};
cout.operator<<(v.operator[](0));
v.operator[](1).operator+=(“!”);
```

examples

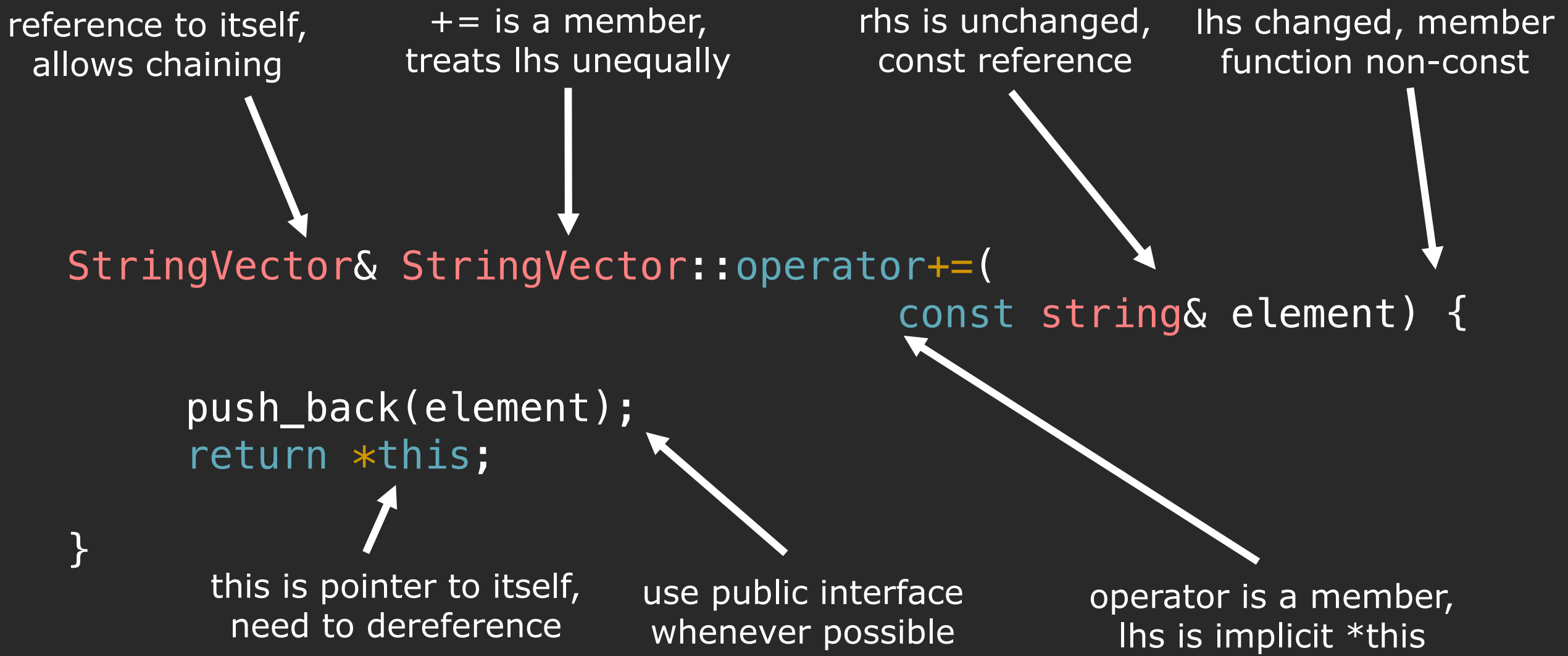
Design considerations when overloading operators!

```
StringVector& StringVector::operator+=(
                                const string& element) {

    push_back(element);
    return *this;

}
```

Design considerations when overloading operators!



Design considerations when overloading operators!

```
StringVector operator+(const StringVector& lhs,
                       const StringVector& rhs) {
    StringVector copy = lhs;
    copy += rhs;
    return copy;

}
```

Design considerations when overloading operators!

returns a copy, separate from l/rhs

+ is nonmember, lhs/rhs treated equally

nonmember, so both lhs/rhs parameters

const reference for both lhs/rhs

```
StringVector operator+(const StringVector& lhs,
                       const StringVector& rhs) {
    StringVector copy = lhs;
    copy += rhs;
    return copy;
}
```

creates copy of lhs

local variable, copied again when returned.

define in terms of other operators to ensure consistency

Member vs. Non-member

MEMBER

1. Must: [], (), ->, =
2. Should: unary operators (++)
3. Both sides not equally treated (+=)

NON-MEMBER

1. lhs is prewritten type (<<)
2. Binary symmetric operator (+, ==, <)
3. Prefer non-friends to friends.

Both const and non-const members declared!

non-const reference, can be written over

must be member

called by non-const objects

```
string& StringVector::operator[](size_t index) {
    // static_cast/const_cast trick
}

const string& StringVector::operator[](size_t index) const {
    return _elems[index];
}
```

const reference cannot be written over

called by const objects

The client could call the subscript for both a const and non-const vector.

```
StringVector v1{“Green”, “Black”, “Oo-long”};

const StringVector v2{“16.9”, “fluid”, “ounces”};

v1[1] = “Hi”; // non-const version, v1[1] is reference
string a = v2[1]; // const version, this works
v2[1] = “Bye”; // not work, v2[1] is const
```

Let’s try overloading the stream insertion operator!

```
ostream& operator<<(ostream& out, const StringVector& sv) {
      out << “{”;
      for (size_t i = 0; i < sv.size(); ++i) {
            out << sv[i];
            if (i != sv.size()-1) out << “, ”;
      }
      out << “}”;
      return os;
}
```

Design considerations for overloading operators.

returns reference, allows for chaining

must be non-member, ostream class not ours!

object inserted is not changed, const ref

```
ostream& operator<<(ostream& out, const StringVector& sv) {
    out << "{";

    // implementation details

    return os;
}
```

writes to the stream

returns reference to stream itself

Questions

Principle of Least Astonishment (POLA)

What do you think it means?

Principle of Least Astonishment (POLA)

"If a necessary feature has a high astonishment factor, it may be necessary to redesign the feature".

Principle of Least Astonishment (POLA)

From the C++ Core Guidelines (section C):

- Design operators primarily to mimic conventional usage.
- Use nonmember functions for symmetric operators.
- Use an operator for an operation with its conventional meaning
- Always provide all out of a set of related operators.

Principle of Least Astonishment (POLA)

```
Time start {15, 30};
Time end {16, 20};
if (start < end) { // obvious
    start += 10; // is this adding to hour or min?
} else {
    end--; // again, hour or min?
    end, 3, 4, 5; // WTF???
}
```

Principle of Least Astonishment (POLA)

Use an operator for an operation with its conventional meaning

- Compound operators return reference to *this
- Arithmetic operators return copies
- In/decrement prefix vs. postfix rules
- Indexing requires const and non-const versions
- Look at the C++ reference for common patterns!

Principle of Least Astonishment (POLA)

```
Fraction a {3, 8};
Fraction b {11, 8};

// equivalent to a.operator+(1), compiles
if (a + 1 == b) cout << “I <3 fractions!”;
// equivalent to 1.operator+(a), does not compile
if (1 + a == b) cout << “I <3 fractions!”;
```

Principle of Least Astonishment (POLA)

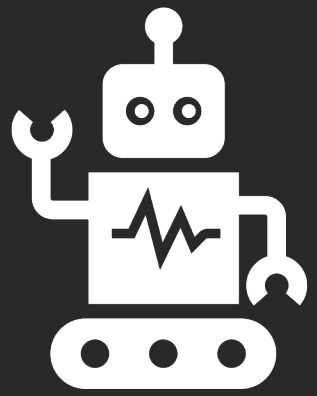
```
Fraction a {3, 8};
Fraction b {9, 16};

// if the following code works
if (a < b) cout << “I <3 fractions!”;
// then the following better work as well
if (b > a) cout << “I <3 fractions!”;
```

Summary of POLA

Operator semantics are very important!

- Member or a non-member?
- Friend or non-friend?
- Const and/or reference of function? Return value? Parameters?
- What is the typical behavior of this operator?



Example

Everything operator overloading

Questions

template classes

Step 1: Add template declaration for class.

```
template <typename T>
    class MyVector
```

Step 2: Add all the member type aliases.

```
using value_type = T;
using size_type = size_t;
using difference_type = ptrdiff_t;
using reference = value_type&;
using const_reference = const value_type&;
using pointer = T*;
using const_pointer = const T*;
```

Step 3: Add the template declaration to every single class member.

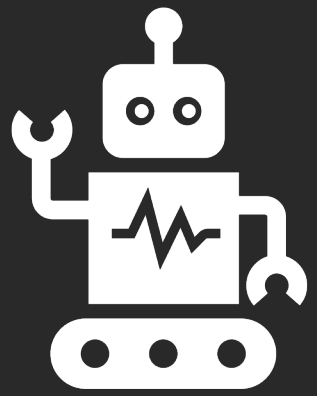
```
template <typename T>
typename StringVector<T>::size_type size() const {
    return _size;
}
```

Step 4: Move everything to the .h file

Annoying reason: separate compilation

template classes are not classes themselves

https://stackoverflow.com/questions/495021/why-can-templates-only-be-implemented-in-the-header-file

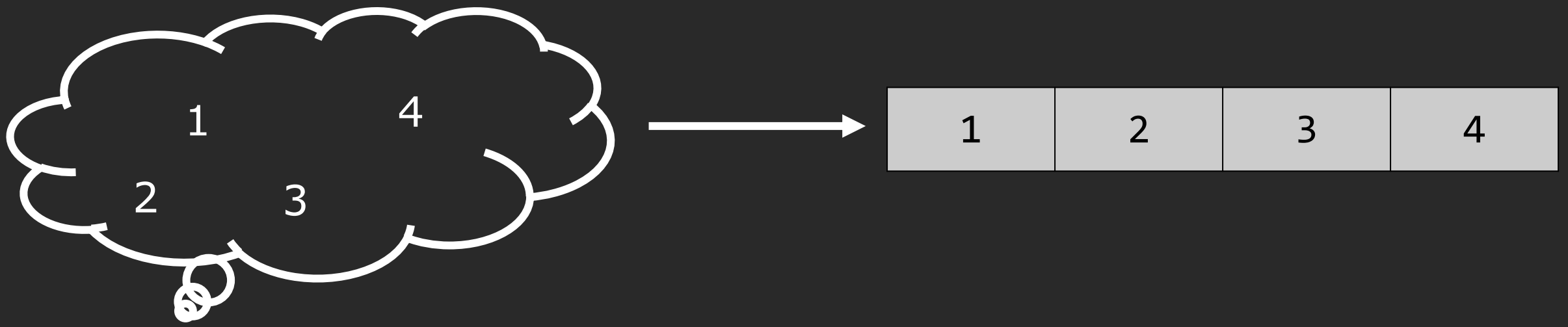


Example

Turning StringVector into MyVector<T>

iterator classes

Iterators allow us to view a non-linear container in a linear manner



Why support iterators?

```
MyVector<string> vec(3, “Hello”);
std::sort(vec.begin(), vec.end());
for (const auto& val : vec) {
    cout << val << ‘\n’;
}
```

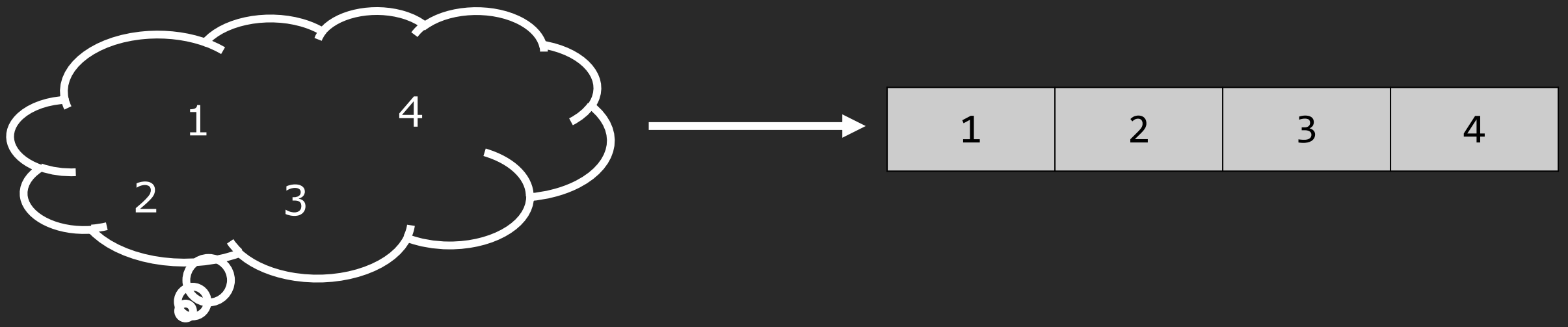
iterators must support these operators

```
auto iter = v.begin();          // copy constructor
copy = iter;                    // copy assignment
if (iter == copy)               // (in)equality
*iter;                          // dereference
++iter;                         // prefix increment
copy++;                         // postfix increment
```

random access iterators are even more powerful

```
iter += 3;                  // compound add
iter + 3;                   // iter + size_t
iter1 - iter2;              // iter - iter
if (iter <= copy)           // comparison
--iter;                     // prefix decrement
copy--;                     // postfix decrement
```

Iterators allow us to view a non-linear container in a linear manner



Iterators

How are they able to represent a non-linear collection in a “sequential” way?

We don’t need to know!

We will just use them like any other thing - assume they just work somehow. This is the power of abstraction!

Iterators

How are they able to represent a non-linear collection in a “sequential” way?

~~We don’t need to know!~~

Not the spirit of C++!

We will just use them like any other thing - assume they just work somehow. This is the power of abstraction!

Design Philosophy of C++

- Allow the programmer full control, responsibility, and choice if they want it.
- Express ideas and intent directly in code.
- Enforce safety at compile time whenever possible.
- Do not waste time or space.
- Compartmentalize messy constructs.

How do you implement an iterator?

You guessed it. More operator overloading!

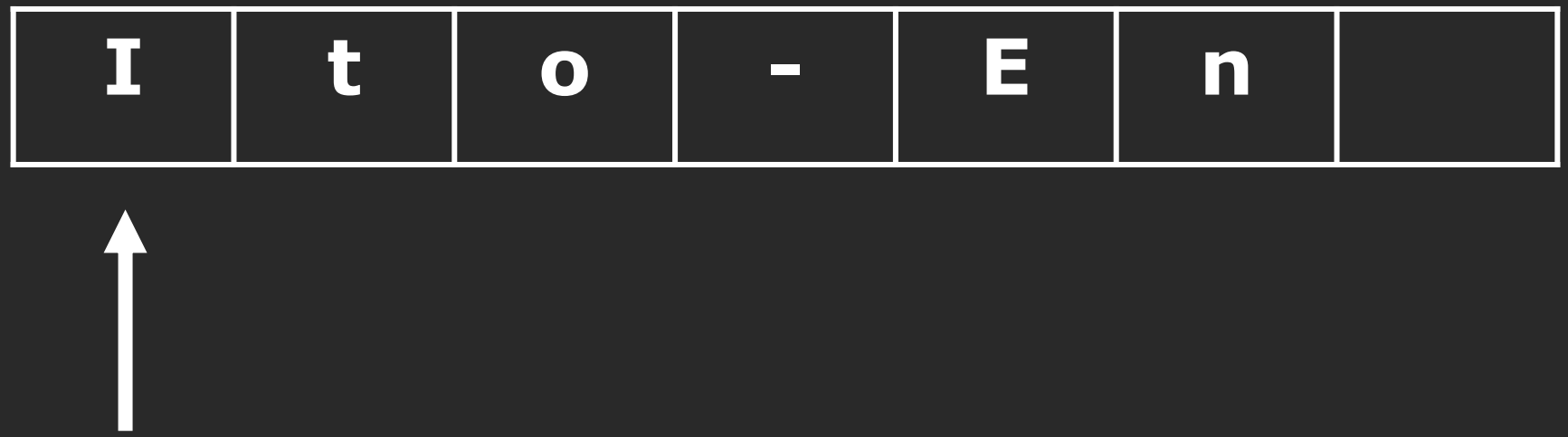
Before we get to that, what are the important steps of class design?

Iterators you've been using…

Let's try to guess how various iterators work!

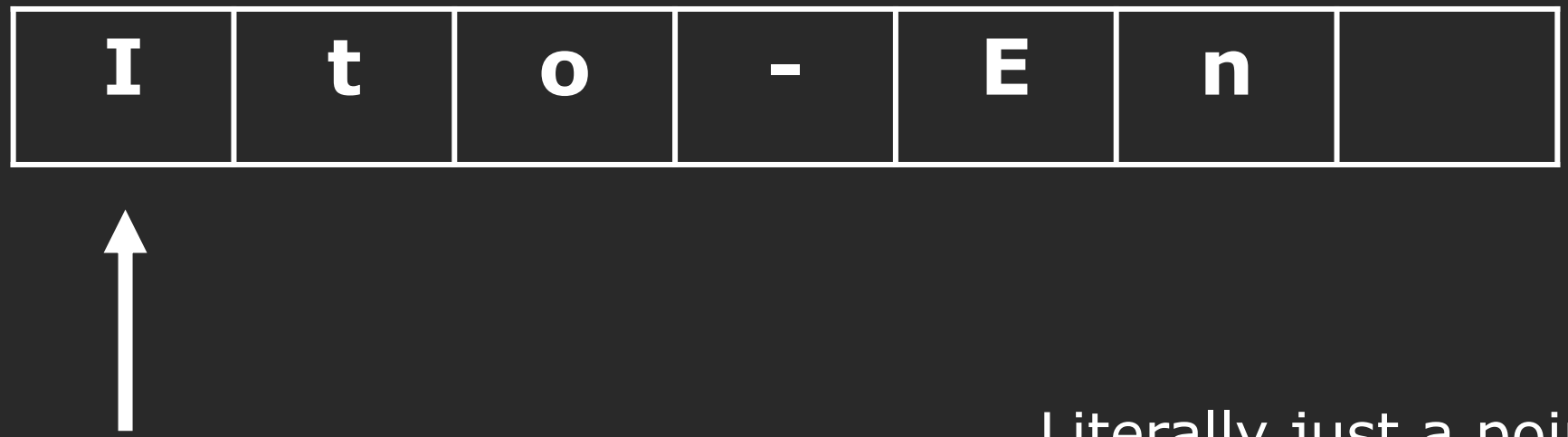
Iterators you've been using…

Let's try to guess how various iterators work!



Iterators you've been using…

Let's try to guess how various iterators work!



Literally just a pointer…

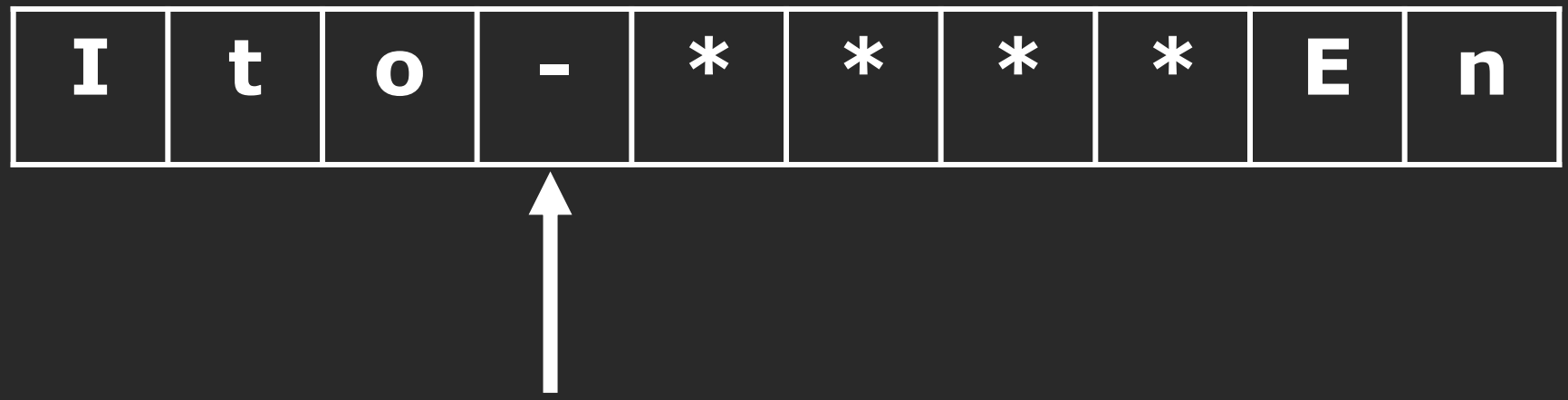
Sidenote: pointers are the iterators for an array.

```
int* arr = new int[10];
auto begin = arr;
auto end = arr + 10;
std::fill(begin, end, 42);
```

pointer knows how large each int is, slides forward 10 integers.

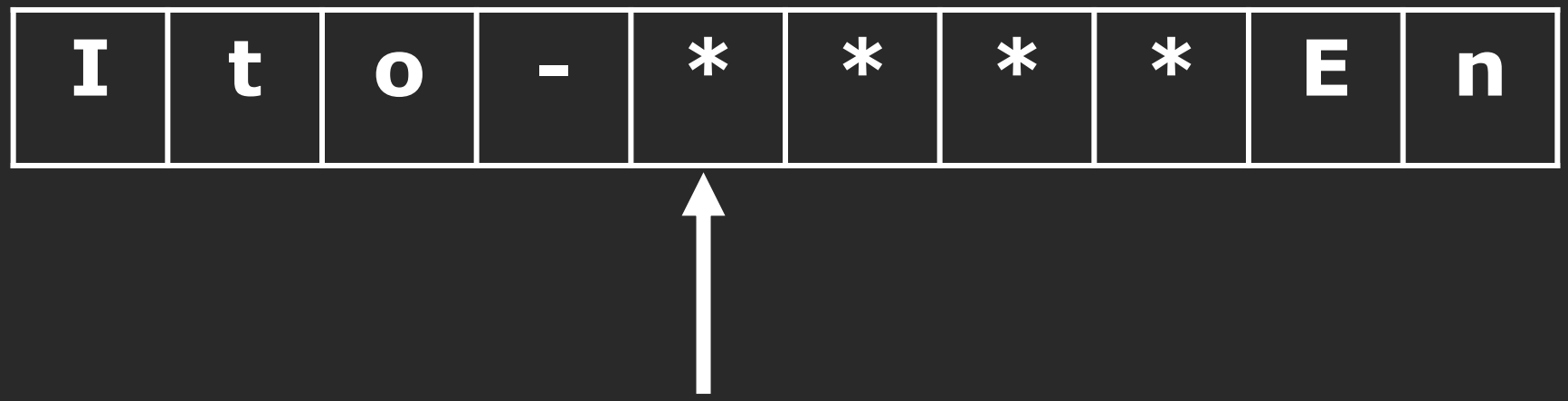
Iterators for a GapBuffer?

Increment?



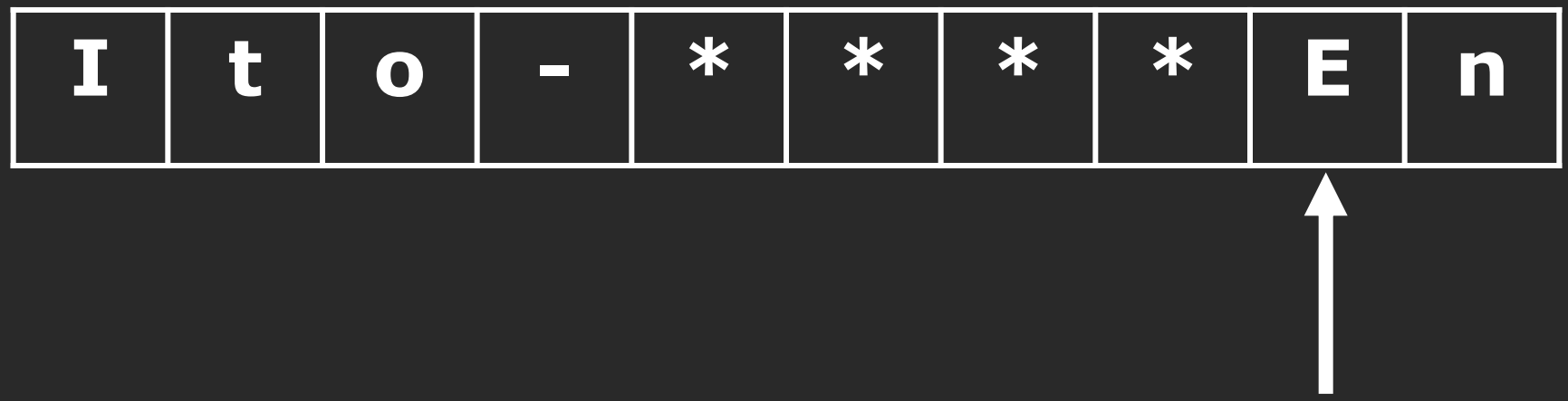
Iterators for a GapBuffer?

If it reaches the gap, needs to jump to the other side!



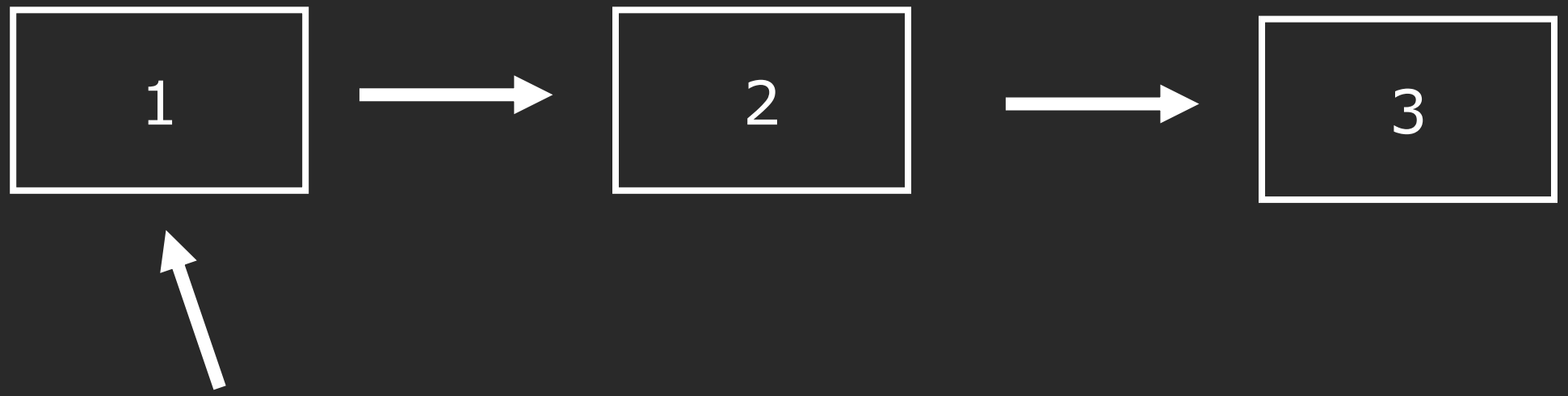
Iterators for a GapBuffer?

If it reaches the gap, needs to jump to the other side!



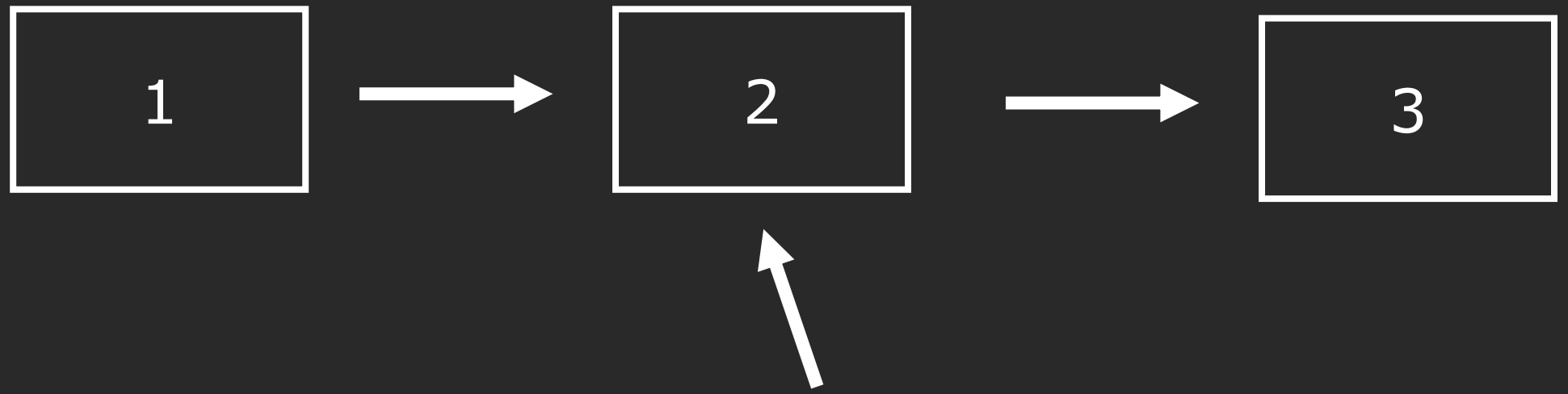
Iterators for a forward_list?

To increment, get the next field of the current node



Iterators for a forward_list?

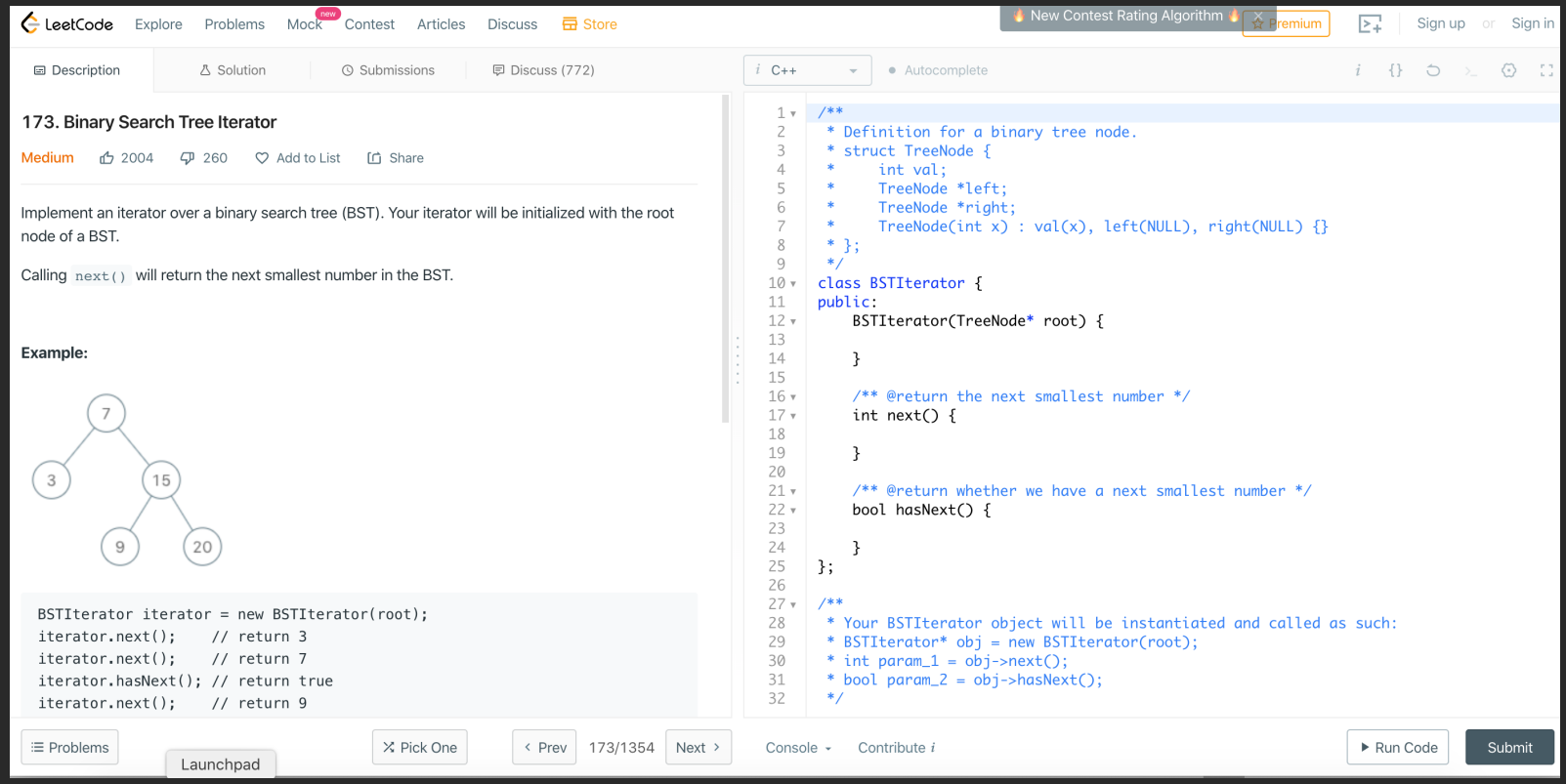
Can you see why forward_list iterators are Forward Iterators, not Bidirectional Iterators?



Iterators for a map/sets?

This is a programming interview question!

(a simplified version is a medium problem on Leetcode)



Design choices for iterators

The iterator should only support operators that it can perform in constant time!

(example: technically you could have + overloaded for list, but the runtime would not be constant!)

Step 1: Determine your private instance (member) variables.

```
T* _pointee;
```

This is a good design! Simple, memory efficient, and intuitive.

Can you think of any downsides of this approach?

Step 1: Determine your private instance (member) variables.

```
MyVector<T>* _pointee;
size_type _index;
```

Point of concession: We want to minimize the dependencies our classes have with each other, and have the iterator store a pointer to the container is not ideal. For StringVector in fact, we really just need a pointer. For GapBuffer, we only need the pointee to a place in the array, index, gap location and size, and capacity. For a linked list in fact, just the pointee to the node! However, this makes implementing iterators very algorithmically challenging. In the interest of keeping things simple, we will break encapsulation and coupling rules.

Step 2: Determine the public behaviors of your class.

- Our class will be a **nested class** (alternative: pimpl)
- Its official name is MyVector::iterator.
- The MyVector and iterator class are mutual friends.
- The iterator class has access to all the member types and template-y stuff.
- The iterator class inherits from an iterator tag.
- A ton of operators.

What is a friend?

- Declaring another function/class as a friend grants them access to the private members of the class.
- Friend class: class can access private members of each.

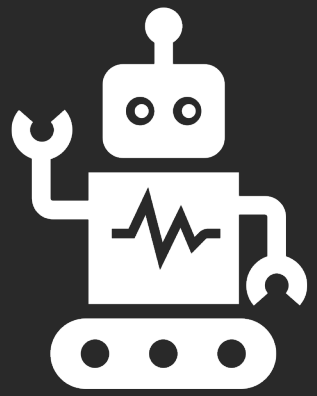
```
friend class iterator;
```

Questions

Step 3: How do we construct an instance of iterator?

- Let’s create a constructor that initializes all members.
- We don’t want the constructor to be accessible by anyone other than the class itself.

- Idea: declare the iterator constructor private.
- MyVector is a friend of iterator, so it can still call the private constructor.



Example

Review the boilerplate code

(it’s not really that important…)

Step 4: implement a ton of operators.

- ++, -- (pre+posfix)
- ==, !=, <, >, <=, >=
- * (deref)
- +=, -=
- +, - (iter+n, n+iter, iter-n, iter-iter)
- = (assignment – next time!)

Prefix vs. Postfix

returns reference to *this

unary operator, implement as member

non-const, since we change iterator position

```
iterator& iterator::operator++();      // prefix

iterator iterator::operator++(int);    // postfix
```

returns a copy of original pointer

used to distinguish between pre/postfix

Comparison operators

```
bool operator<(const iterator& lhs,
               const iterator& rhs) {
    // you implement!
}

bool operator>(const iterator& lhs,
               const iterator& rhs) {

}
```

Comparison operators

```
bool operator<(const iterator& lhs,
               const iterator& rhs) {
    // you implement!
}

bool operator>(const iterator& lhs,
               const iterator& rhs) {
    return !(lhs < rhs);
}
```

Comparison operators

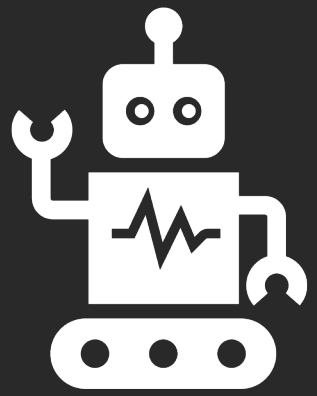
- Define != in terms of ==
- Define >, <=, >= in terms of == and <

- Why? Ensures consistency of operators.
- Is it annoying we have to write these every time? C++20 introduces spaceship operator.

Step 5: add iterator creators in MyVector

```
iterator begin() {
    return iterator(this, 0);
}

iterator end() {
    return iterator(this, _logical_size);
}
```



Example

Implementing operators +, +=, ==, <, etc for MyVector

Note about GapBuffer part 5

- You only have to implement *, ++, --, +, and -.
- (the code for these operators in GapBuffer is the same as the ones in MyVector, which is why we didn’t write it)

- We implemented all the friend operators for you.
- Read the code, think about const-ness, member vs. non-member, etc.

Note about GapBuffer part 5

- Technically you need a const_iterator class, which is an iterator pointing to a const GapBuffer.
- Seems unnecessarily tedious, so you don't have to do it. If you do however, the stl library will work nicer! ☺

Questions

looking ahead

There are a few more interesting operators.

Arithmetic	+	-	*	/	%		
	+=	-=	*=	/=	%=		
Bitwise		&	|	~	!		
Relational	==	!=	<	>	<=	>=	<=>
Stream	<<	>>	<<=	>>=			
Logical	&&	||	^	&=	|=	^=	
Increment	++	--					
Memory	->	->*	new	new []	delete	delete []	
Misc	( )	[ ]	,	=		co_await	

Automatic memory management: smart pointers (lecture 17)

```
unique_pointer<Node> ptr{new Node(0)}
ptr->next = nullptr;
```

Functors (lecture 7 – lambdas)

```
class GreaterThan {
  public:
    GreaterThan(int limit) : limit(limit) {}
    bool operator() (int val) {return val >= limit};
  private:
    int limit;
}
int main() {
  int limit = getInteger("Minimum for A?");
  vector<int> grades = readStudentGrades();
  GreaterThan func(limit);
  cout << countOccurences(pi.begin(), pi.end(), func);
}
```

You can define your own memory allocators!

operator new, operator new[]

Defined in header <new>

replaceable allocation functions
[[nodiscard]] (since C++20)

```
void* operator new  ( std::size_t count );                                  (1)
void* operator new[]( std::size_t count );                                  (2)
void* operator new  ( std::size_t count, std::align_val_t al);              (3) (since C++17)
void* operator new[]( std::size_t count, std::align_val_t al);              (4) (since C++17)
```

replaceable non-throwing allocation functions
[[nodiscard]] (since C++20)

```
void* operator new  ( std::size_t count, const std::nothrow_t& tag);        (5)
void* operator new[]( std::size_t count, const std::nothrow_t& tag);        (6)
void* operator new  ( std::size_t count,
                      std::align_val_t al, const std::nothrow_t&);          (7) (since C++17)
void* operator new[]( std::size_t count,
                      std::align_val_t al, const std::nothrow_t&);          (8) (since C++17)
```

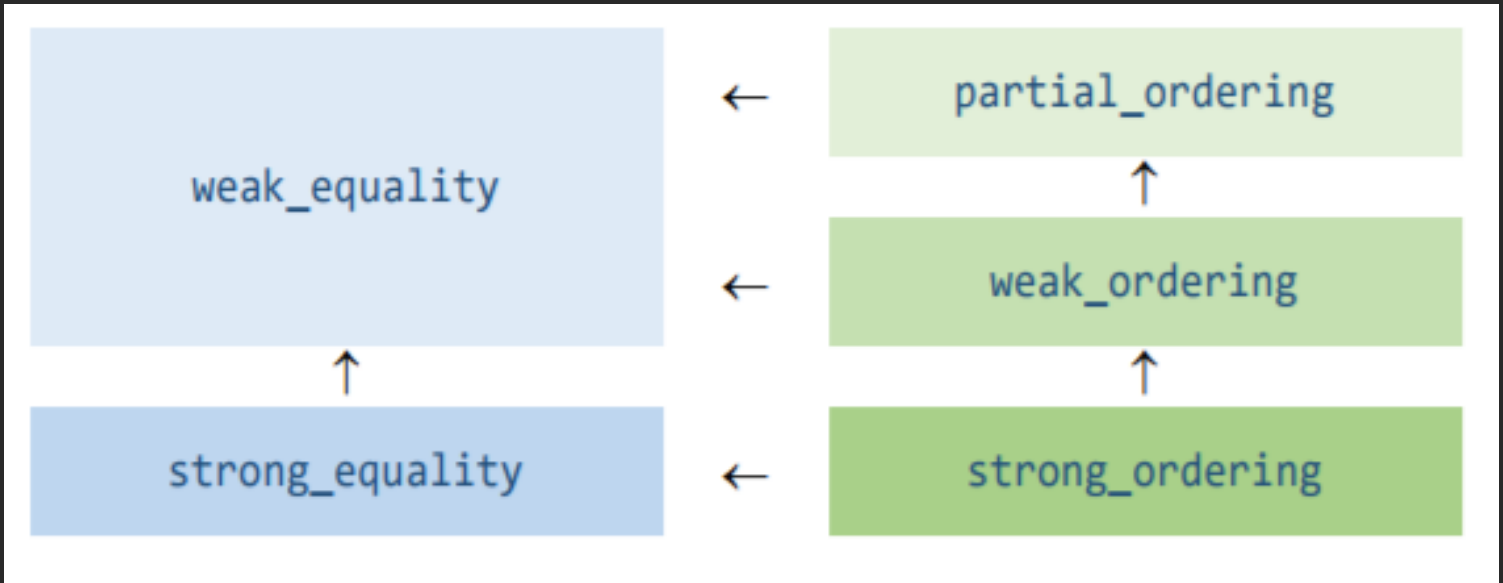
non-allocating placement allocation functions
[[nodiscard]] (since C++20)

```
void* operator new  ( std::size_t count, void* ptr );                       (9)
void* operator new[]( std::size_t count, void* ptr );                       (10)
```

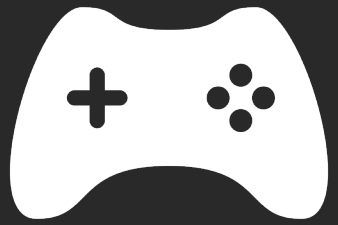
Advanced Multithreading Support (C++20)

```
awaiter operator co_await() const noexcept {
    return awaiter{ *this };
}
```

Spaceship operator (C++20)




```
std::strong_ordering operator<=> (const Time& rhs) {
    return hour <=> rhs.hour;
}
```



Next time

Special Member Functions