

Advanced Containers and Iterators

Bad Dad Joke of the Day:

- Dad: Hey son, take some of this jerky 🤔*
- Son: Thanks dad 😊
- Dad: It was nice meating you

*emojis replicated as given

Creds: Adam

Game Plan



- Map Iterators
- Further Iterator Usages
- Announcements
- Iterator Types

Brief Recap

Associative Containers

Useful abstraction for “**associating**” a key with a value.

```
std::map
```

```
map<string, int> directory;    // name -> phone number
```

```
std::set
```

```
set<string> dict;              // does it contain a word?
```

Iterators provide a guaranteed interface!

Four essential iterator operations:

Create iterator

Dereference iterator to read value currently pointed to

Advance iterator

Compare against another iterator (especially `.end()` iterator)

Iterators provide a guaranteed interface!

Four essential iterator operations:

Create iterator

```
std::set<int>::iterator iter = mySet.begin();
```

Dereference iterator to read value currently pointed to

Advance iterator

Compare against another iterator (especially `.end()` iterator)

Iterators provide a guaranteed interface!

Four essential iterator operations:

Create iterator

```
std::set<int>::iterator iter = mySet.begin();
```

Dereference iterator to read value currently pointed to

```
int val = *iter;
```

Advance iterator

Compare against another iterator (especially `.end()` iterator)

Iterators provide a guaranteed interface!

Four essential iterator operations:

Create iterator

```
std::set<int>::iterator iter = mySet.begin();
```

Dereference iterator to read value currently pointed to

```
int val = *iter;
```

Advance iterator

```
iter++; or ++iter;
```

Compare against another iterator (especially `.end()` iterator)

Iterators provide a guaranteed interface!

Four essential iterator operations:

Create iterator

```
std::set<int>::iterator iter = mySet.begin();
```

Dereference iterator to read value currently pointed to

```
int val = *iter;
```

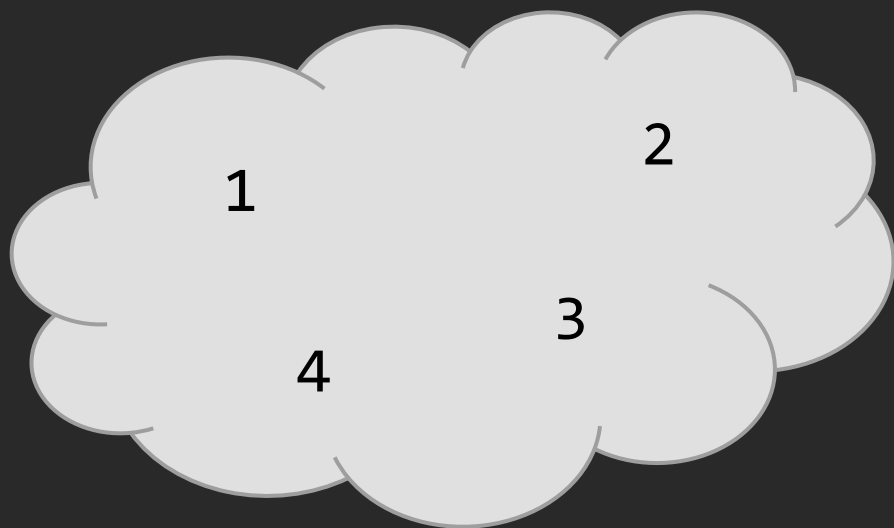
Advance iterator

```
iter++; or ++iter;
```

Compare against another iterator (especially `.end()` iterator)

```
if (iter == mySet.end()) return;
```

The Result

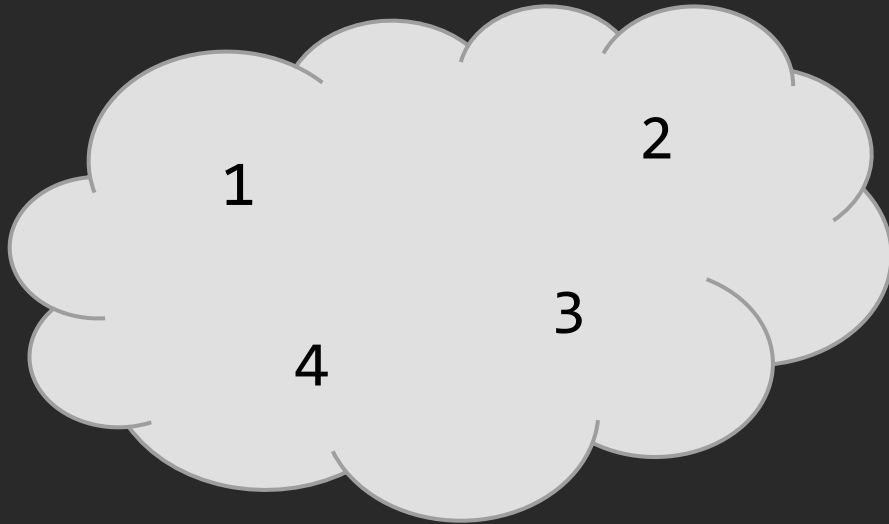


The Result

This could be a

```
std::vector<Node> myVec,
```

```
std::set<int> mySet, etc.
```



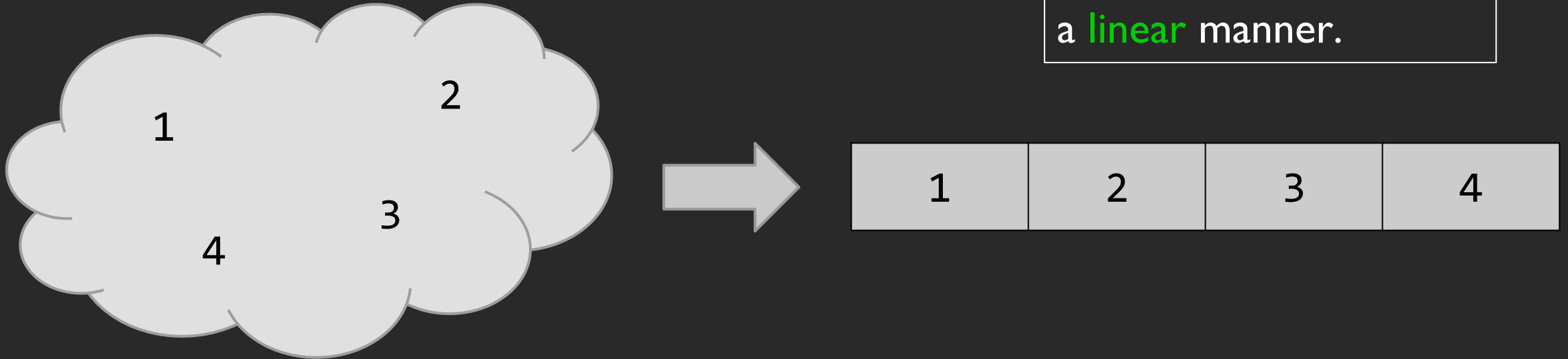
The Result

This could be a

```
std::vector<Node> myVec,
```

```
std::set<int> mySet, etc.
```

Iterators let us view a
non-linear collection in
a **linear** manner.



The Result

The Result

This means that we can use the exact same code to perform a logical action, *regardless of the data structure!*

The Result

This means that we can use the exact same code to perform a logical action, *regardless of the data structure*!

```
int numOccurrences(vector<int>& cont, int elemToCount) {  
    int counter = 0;  
    vector<int>::iterator iter;  
    for(iter = cont.begin(); iter != cont.end(); ++iter) {  
        if(*iter == elemToCount)  
            ++counter;  
    }  
    return counter;  
}
```

The Result

This means that we can use the exact same code to perform a logical action, *regardless of the data structure*!

```
int numOccurrences(list<int>& cont, int elemToCount) {  
    int counter = 0;  
    list<int>::iterator iter;  
    for(iter = cont.begin(); iter != cont.end(); ++iter) {  
        if(*iter == elemToCount)  
            ++counter;  
    }  
    return counter;  
}
```


The Result

This means that we can use the exact same code to perform a logical action, *regardless of the data structure*!

```
int numOccurrences(set<int>& cont, int elemToCount) {  
    int counter = 0;  
    set<int>::iterator iter;  
    for(iter = cont.begin(); iter != cont.end(); ++iter) {  
        if(*iter == elemToCount)  
            ++counter;  
    }  
    return counter;  
}
```

The Result

This means that we can use the exact same code to perform a logical action, *regardless of the data structure*!

```
int numOccurrences(???& cont, ??? elemToCount) {  
    int counter = 0;  
    ???::iterator iter;  
    for(iter = cont.begin(); iter != cont.end(); ++iter) {  
        if(*iter == elemToCount)  
            ++counter;  
    }  
    return counter;  
}
```

Map Iterators

Aside: The `std::pair` Class

A pair is simply two objects bundled together.

Syntax:

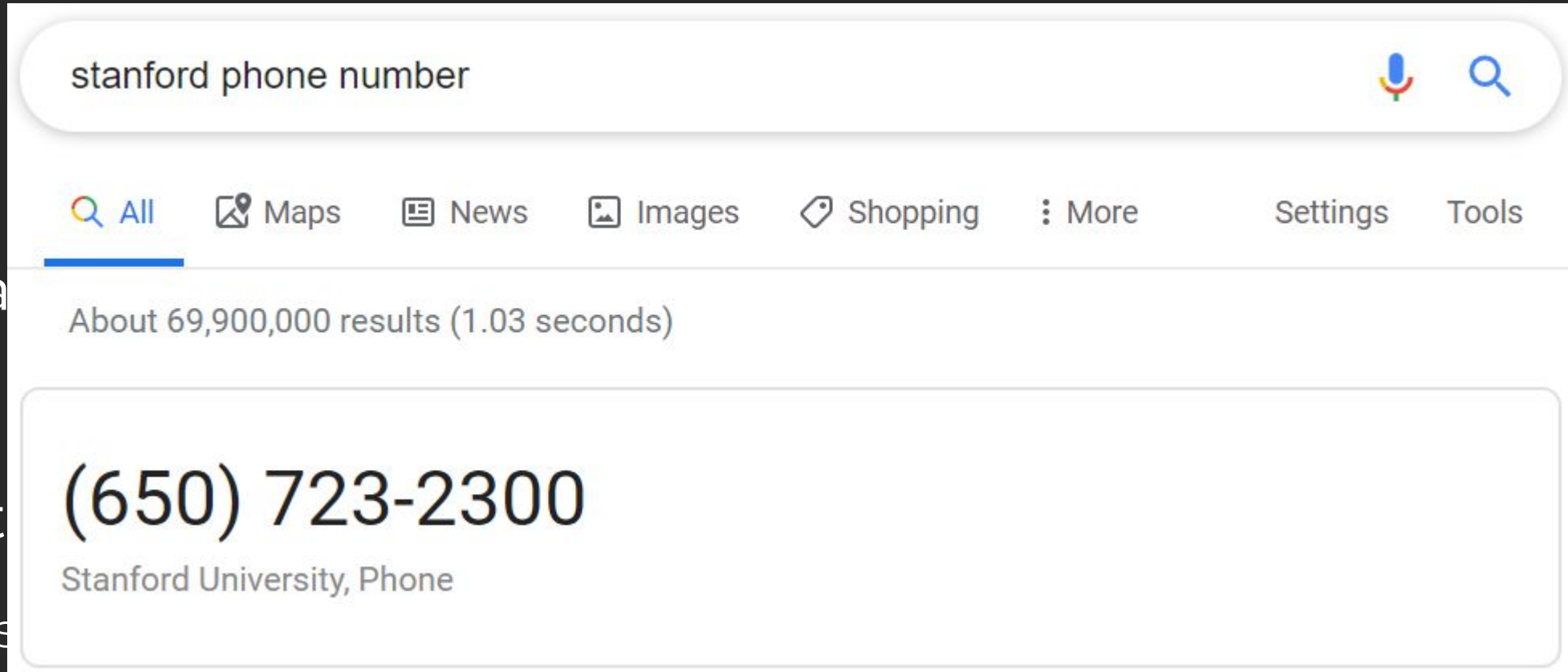
```
std::pair<string, int> p;
```

```
p.first = "Phone number";
```

```
p.second = 6507232300;
```

A pa

Synt



```
p.first = "Phone number";
```

```
p.second = 6507232300;
```

Aside: The `std::pair` Class

A pair is simply two objects bundled together.

Syntax:

```
std::pair<string, int> p;
```

```
p.first = "Phone number";
```

```
p.second = 6507232300;
```

Aside: The `std::pair` Class

Quicker ways to make a pair:

```
std::pair<string, int> p{"Phone number", 6507232300};  
std::make_pair("Phone number", 6507232300);
```

What's the difference?

Aside: The `std::pair` Class

Quicker ways to make a pair:

```
std::pair<string, int> p{"Phone number", 6507232300};  
std::make_pair("Phone number", 6507232300);
```

What's the difference?

`make_pair` **automatically deduces the type!**

This is a great place to use `auto`!

```
auto time = std::make_pair(1, 45);
```


Example: Multimaps

Recall: `std::multimap` is a map that permits multiple entries with the same key.

Doesn't have `[]` operator!

Example: Multimaps

Recall: `std::multimap` is a map that permits multiple entries with the same key.

Doesn't have `[]` operator!

Instead, add elements by calling `.insert` on a key value `std::pair`.

```
std::multimap<int, int> myMMap;  
myMMap.insert(make_pair(3, 3));  
myMMap.insert({3, 12}); // shorter syntax  
cout << myMMap.count(3) << endl; // prints 2
```

Map Iterators

Map iterators are slightly different because we have both keys and values.

Dereferencing a `set<string>` iterator gives you a `string`.

Dereferencing a `map<string, int>` iterator gives you an `std::pair<string, int>`.

Map Iterators

Example:

```
map<int, int> m;  
map<int, int>::iterator i = m.begin();  
map<int, int>::iterator end = m.end();  
while (i != end) {  
    cout << (*i).first << (*i).second << endl;  
    ++i;  
}
```

Map Iterators

Example:

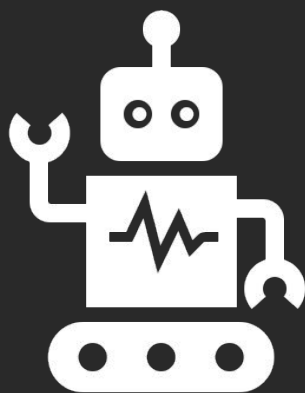
```
map<int, int> m;  
map<int, int>::iterator i = m.begin();  
map<int, int>::iterator end = m.end();  
while (i != end) {  
    cout << (*i).first << (*i).second << endl;  
    ++i;  
}
```

Further Iterator Usages

Iterator Uses

Iterators are useful for more than just looping through an entire container!

Let's take a look...



Example

Iterator Uses

Iterator Uses - Sorting

For example, we sorted a vector using

```
std::sort(vec.begin(), vec.end());
```

Iterator Uses - Find

Finding elements

```
const int elemToFind = 5;
vector<int>::iterator it = std::find(vec.begin(),
                                    vec.end(), elemToFind);

if(it != vec.end()) {
    cout << "Found: " << *it << endl;
} else {
    cout << "Element not found!" << endl;
}
```

Aside: find vs. count

If you recall from last lecture, associative containers have a method called `count(key)`

- Equivalent of Stanford `myMap.containsKey(key)`:
 - `myMap.count(key)`
 - `if (myMap.count(key) == 0) cout << "Not Found";`

Aside: find vs. count

If you recall from last lecture, associative containers have a method called `count(key)`

- Equivalent of Stanford `myMap.containskey(key)`:
 - `myMap.count(key)`
 - `if (myMap.count(key) == 0) cout << "Not Found";`
 - `std::find(myMap.begin(), myMap.end(), key);`
 - `if (find(myMap.begin(), myMap.end(), key) == myMap.end())
cout << "Not Found";`

Aside: find vs. count

If you recall from last lecture, associative containers have a method called `count(key)`

- Equivalent of Stanford `myMap.containsKey(key)`:
 - `myMap.count(key)`
 - `std::find(myMap.begin(), myMap.end(), key);`

Aside: find vs. count

If you recall from last lecture, associative containers have a method called `count(key)`

- Equivalent of Stanford `myMap.containskey(key)`:

- `myMap.count(key)`
- `std::find(myMap.begin(), myMap.end(), key);`



- `count` is actually just a call to the `find` function! So `find` is marginally faster

Of course, C++20 will bring a new `contains(key)` method... but until then, use `find`.

Iterator Uses - Ranges

Finding elements

```
set<int>::iterator iter = mySet.lower_bound(7);  
set<int>::iterator end = mySet.lower_bound(26);  
while (iter != end) {  
    cout << *i << endl;  
    ++i;  
}
```

Iterator Uses - Ranges

We can iterate through different ranges

	[a, b]	[a, b)	(a, b]	(a, b)
begin	lower_bound(a)	lower_bound(a)	upper_bound(a)	upper_bound(a)
end	upper_bound(b)	lower_bound(b)	upper_bound(b)	lower_bound(b)

Range Based `for` Loop

```
map<string, int> myMap;  
for(auto thing : myMap) {  
    doSomething(thing.first, thing.second);  
}
```

Range Based `for` Loop

A range based `for` loop is (more or less) a shorthand for iterator code:

```
map<string, int> myMap;  
for(auto thing : myMap) {  
    doSomething(thing.first, thing.second);  
}
```



```
map<string, int> myMap;  
for(auto iter = myMap.begin(); iter != myMap.end(); ++iter) {  
    auto thing = *iter;  
    doSomething(thing.first, thing.second);  
}
```

Range Based `for` Loop

A range based `for` loop is (more or less) a shorthand for iterator code:

6.5.4 The range-based `for` statement

[stmt.ranged]

- 1 For a range-based `for` statement of the form

`for (for-range-declaration : expression) statement`

let *range-init* be equivalent to the *expression* surrounded by parentheses⁸⁶

`(expression)`

and for a range-based `for` statement of the form

`for (for-range-declaration : braced-init-list) statement`

let *range-init* be equivalent to the *braced-init-list*. In each case, a range-based `for` statement is equivalent to

```
{  
    auto && __range = range-init;  
    for ( auto __begin = begin-expr,  
          __end = end-expr;  
          __begin != __end;  
          ++__begin ) {  
        for-range-declaration = *__begin;  
        statement  
    }
```

Announcements

Announcements

- Office hours for Assignment 1 on Piazza!
- Feedback form #1 released! Fill out by next Thursday (1/30) for an extra late day.

<https://bit.ly/2vfV4As>

Quick Review of Structs

for Assignment 1!

General Struct Syntax

```
// Declaring the struct definition
```

```
struct Object {  
    type var1;  
    type var2;  
}
```

```
// Initializing a struct object using uniform initialization
```

```
struct Object objName{value1, value2};  
    // "struct" keyword is optional in C++
```

```
// Operating on the struct object - in this case, assigning a  
value
```

```
objName.var1 = newvalue1;
```

General Struct Syntax

```
struct SimpleGraph {  
    vector<Node> nodes;  
    vector<Edge> edges;  
}
```

```
struct Node {  
    double x;  
    double y;  
}
```


General Struct Syntax

```
struct SimpleGraph {  
    vector<Node> nodes;  
    vector<Edge> edges;  
}
```

```
struct Node {  
    double x;  
    double y;  
}
```

```
struct SimpleGraph graph{};  
// How would you add a Node to the graph?
```

General Struct Syntax

```
struct SimpleGraph {  
    vector<Node> nodes;  
    vector<Edge> edges;  
}
```

```
struct Node {  
    double x;  
    double y;  
}
```

```
struct SimpleGraph graph{};
```

```
// How would you add a Node to the graph?
```

```
graph.nodes.push_back( {someXValue, someYValue} );
```

```
    // automatically creates Node object + adds to vector
```

Iterator Types

Iterator Types

So far we have only really incremented iterators.

But for some containers, we should be able to do things like:

```
std::vector<int> v(10);  
auto mid = v.begin() + v.size()/2;
```

```
std::deque<int> d(13);  
auto some_iter = d.begin() + 3;
```

Iterator Types

So far we have only really incremented iterators.

But for some containers, we should be able to do things like:

```
std::vector<int> v(10);  
auto mid = v.begin() + v.size()/2;
```

```
std::deque<int> d(13);  
auto some_iter = d.begin() + 3;
```

Seems right!

Iterator Types

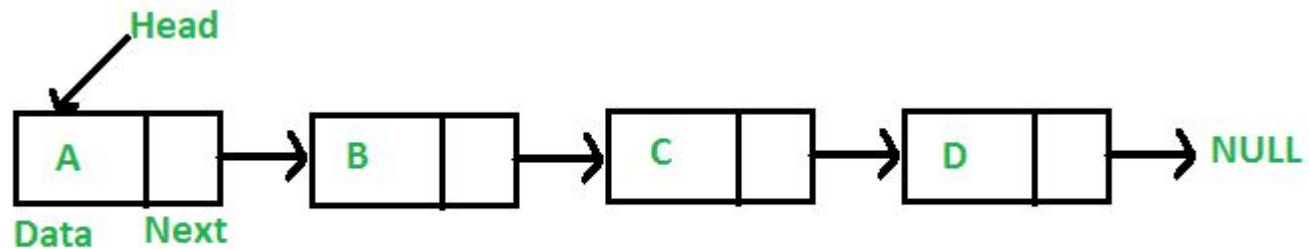
But what about `std::list` (doubly linked list)?

```
std::list<int> myList(10);  
auto some_iter = myList.begin() + 3;
```

Iterator Types

But what about `std::list` (doubly linked list)?

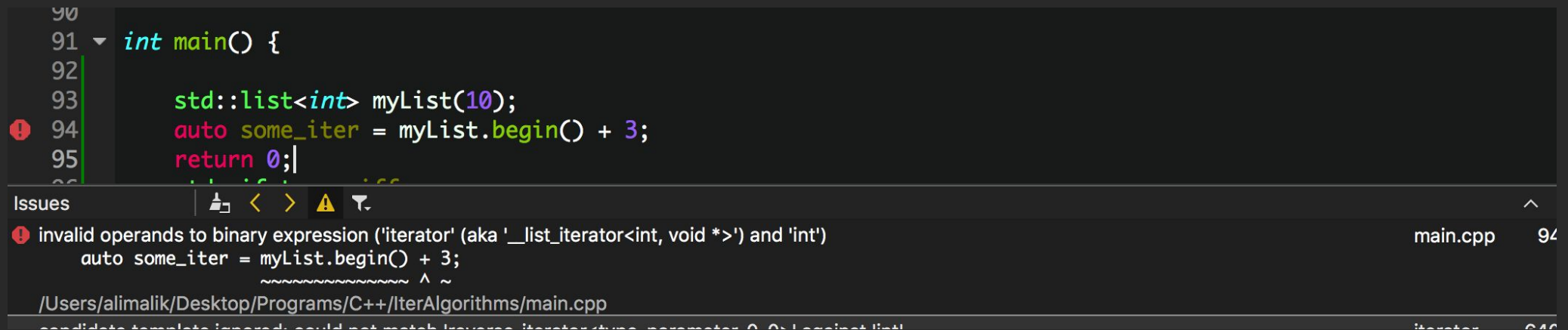
```
std::list<int> myList(10);  
auto some_iter = myList.begin() + 3;
```



Iterator Types

But what about `std::list` (doubly linked list)?

```
std::list<int> myList(10);  
auto some_iter = myList.begin() + 3;
```



```
90  
91 int main() {  
92  
93     std::list<int> myList(10);  
94     auto some_iter = myList.begin() + 3;  
95     return 0;  
96 }
```

Issues

- invalid operands to binary expression ('iterator' (aka '`__list_iterator<int, void*>`') and '`int`')
auto some_iter = myList.begin() + 3;
~~~~~ ^ ~

/Users/alimalik/Desktop/Programs/C++/IterAlgorithms/main.cpp

candidate template ignored: could not match `__list_iterator` parameter 2: `int` against `list`



# Iterator Types

But what about `std::list` (doubly linked list)?

```
std::list<int> myList(10);  
auto some_iter = myList.begin() + 3;
```

# Iterator Types

But what about `std::list` (doubly linked list)?

What's going on here?

```
std::list<int> myList(10);  
auto some_iter = myList.begin() + 3;
```

```
90  
91 ▾ int main() {  
92  
93     std::list<int> myList(10);  
94     auto some_iter = myList.begin() + 3;  
95     return 0;  
96 }
```

Issues

- invalid operands to binary expression ('iterator' (aka '`__list_iterator<int, void*>`') and '`int`')  
auto some\_iter = myList.begin() + 3;  
~~~~~ ^ ~

/Users/alimalik/Desktop/Programs/C++/IterAlgorithms/main.cpp

Iterator Types

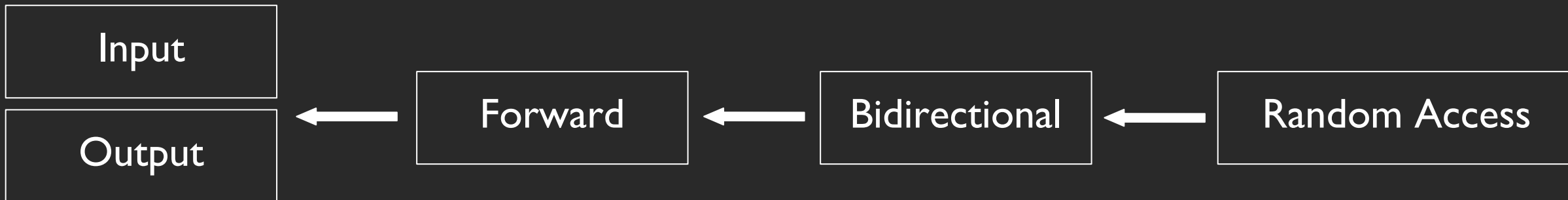
There are 5 different types of iterators!

1. Input
2. Output
3. Forward
4. Bidirectional
5. Random access

Iterator Types

There are 5 different types of iterators!

1. Input
2. Output
3. Forward
4. Bidirectional
5. Random access



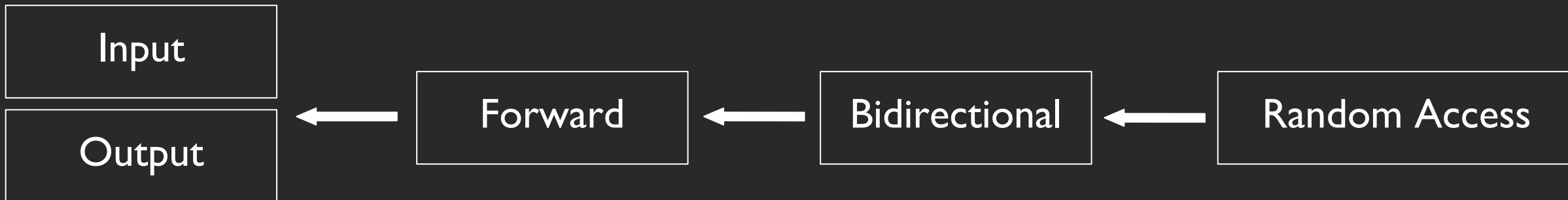
Iterator Types - Similarities

All iterators share a few **common** traits:

Can be created from existing iterator

Can be advanced using **++**

Can be compared with **==** and **!=**

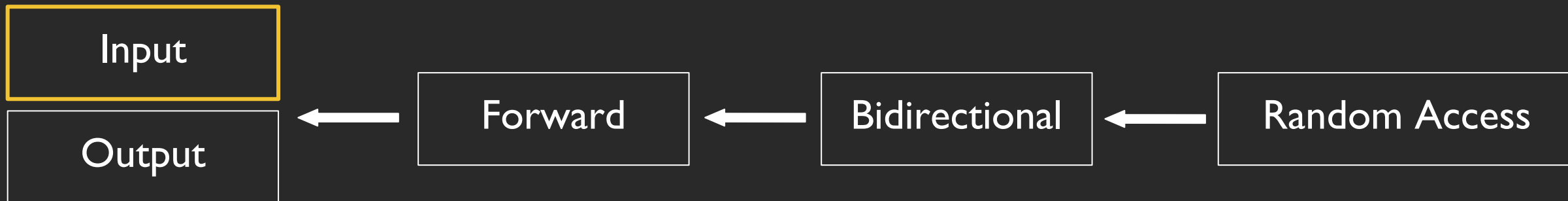


Input Iterators

For sequential, **single-pass** **input**.

Read only i.e. can only be dereferenced on **right** side of expression.

```
vector<int> v = ...  
vector<int>::iterator itr = v.begin();  
int val = *itr;
```



Input Iterators

For sequential, **single-pass** input.

Read only i.e. can only be dereferenced on **right** side of expression.

```
vector<int> v = ...  
vector<int>::iterator itr = v.begin();  
int val = *itr;
```

We've seen these already!

```
template< class InputIt, class T >  
InputIt find( InputIt first, InputIt last, const T& value );
```

```
template< class InputIt, class T >  
typename iterator_traits<InputIt>::difference_type  
count( InputIt first, InputIt last, const T &value );
```

Input Iterators

For sequential, **single-pass** input.

Read only i.e. can only be dereferenced on **right** side of expression.

```
vector<int> v = ...  
vector<int>::iterator itr = v.begin();  
int val = *itr;
```

We've seen these already!

```
template< class InputIt, class T >  
InputIt find( InputIt first, InputIt last, const T& value );
```

```
template< class InputIt, class T >  
typename iterator_traits<InputIt>::difference_type  
count( InputIt first, InputIt last, const T &value );
```



Use cases:

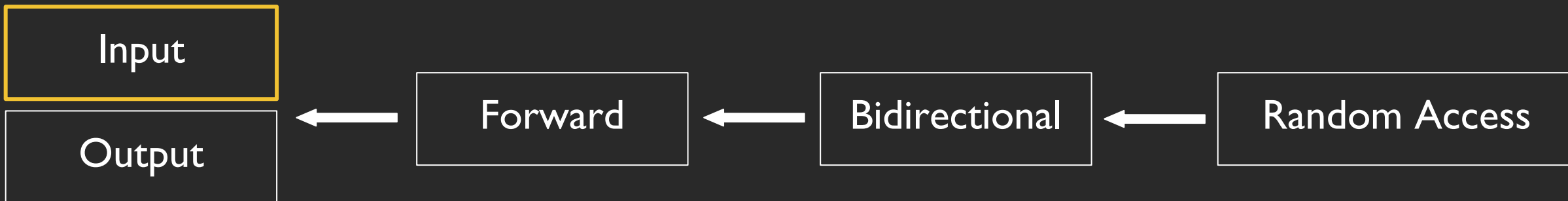
- find and count
- input streams

Input Iterators

For sequential, **single-pass** **input**.

Read only i.e. can only be dereferenced on **right** side of expression.

```
vector<int> v = ...  
vector<int>::iterator itr = v.begin();  
int val = *itr;
```

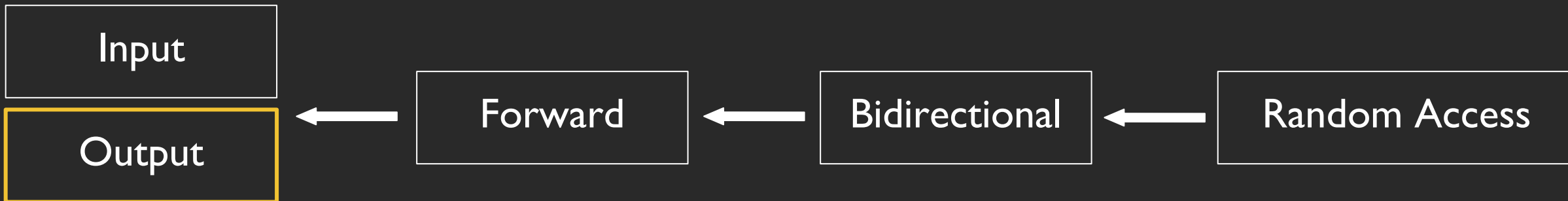


Output Iterators

For sequential, **single-pass** output.

Write only i.e. can only be dereferenced on **left** side of expression.

```
vector<int> v = ...  
vector<int>::iterator itr = v.begin();  
*itr = 12;
```



Output Iterators

For sequential, **single-pass** output.

Write only i.e. can only be dereferenced on **left** side of expression.

```
vector<int> v = ...  
vector<int>::iterator itr = v.begin();  
*itr = 12;
```

Use cases:

- copy:

```
template< class InputIt, class OutputIt >  
OutputIt copy( InputIt first, InputIt last, OutputIt d_first );
```

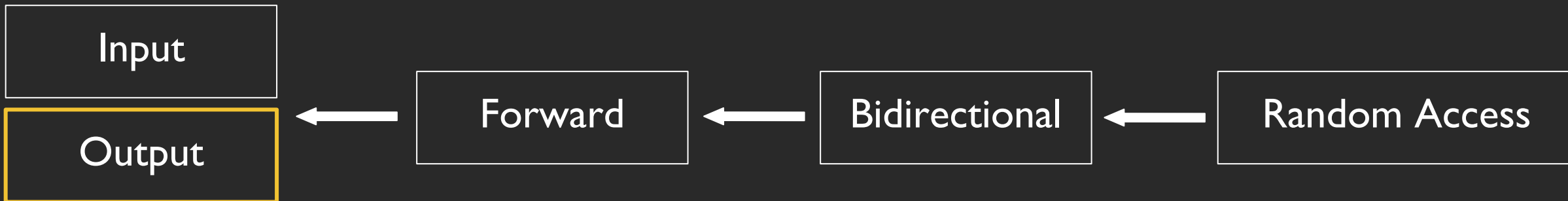
- output streams

Output Iterators

For sequential, **single-pass** output.

Write only i.e. can only be dereferenced on **left** side of expression.

```
vector<int> v = ...  
vector<int>::iterator itr = v.begin();  
*itr = 12;
```

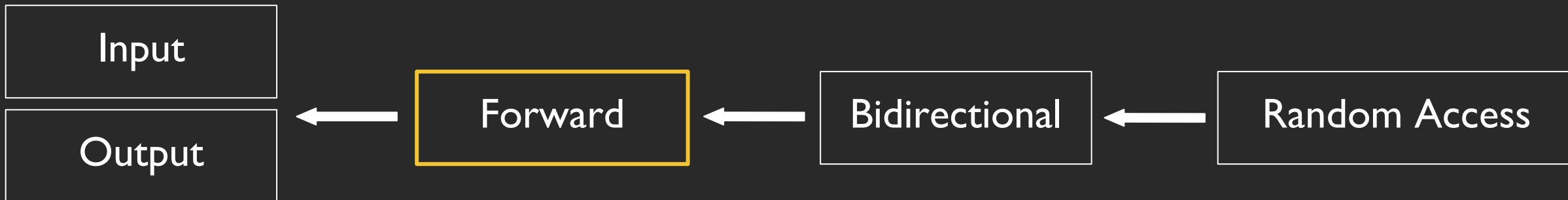


Forward Iterators

Combines input and output iterators, + can make **multiple** passes.

Can read from and write to (if not **const** iterator).

```
vector<int> v = ...  
vector<int>::iterator itr = v.begin();  
int val = *itr;  
*itr = 12;
```



Forward Iterators

Combines input and output iterators, + can make **multiple** passes.

Can read from and write to (if not **const** iterator).

```
vector<int> v = ...  
vector<int>::iterator itr = v.begin();  
int val = *itr;  
*itr = 12;
```

Use cases:

- replace:

```
template< class ForwardIt, class T >  
void replace( ForwardIt first, ForwardIt last,  
              const T& old_value, const T& new_value );
```

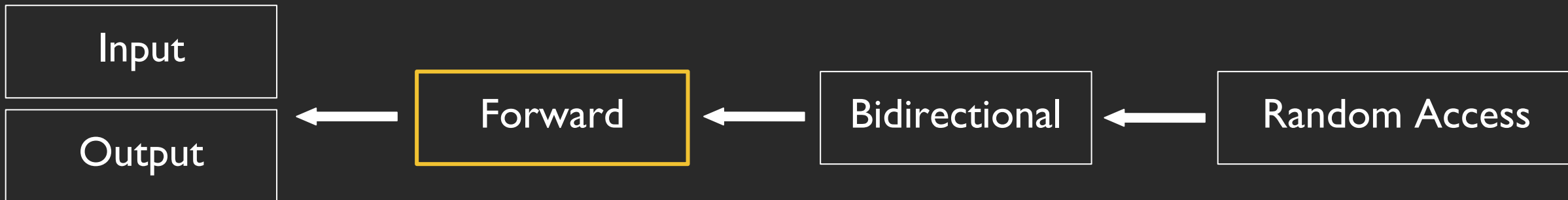
- std::forward_list (sequence container, think of as singly-linked list)

Forward Iterators

Combines input and output iterators, + can make **multiple** passes.

Can read from and write to (if not **const** iterator).

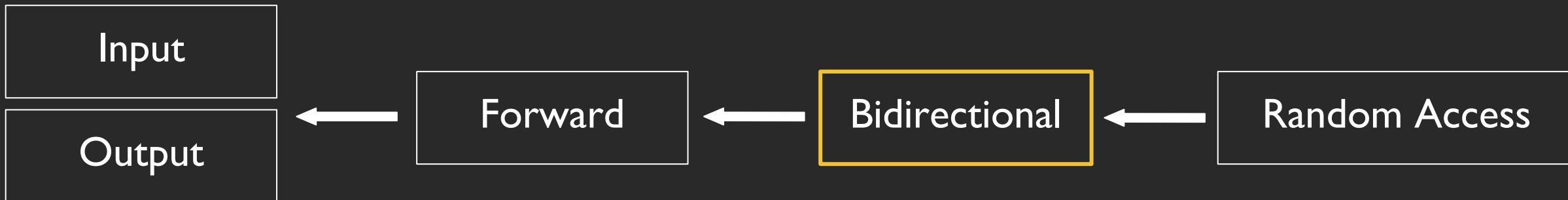
```
vector<int> v = ...  
vector<int>::iterator itr = v.begin();  
int val = *itr;  
*itr = 12;
```



Bidirectional Iterators

Same as forward iterators, + can go backwards with the decrement operator (`--`) .

```
vector<int> v = ...  
vector<int>::iterator itr = v.begin();  
++itr;  
int val = *itr;  
--itr;  
int val2 = *itr;
```



Bidirectional Iterators

Same as forward iterators, + can go backwards with the decrement operator (--) .

```
vector<int> v = ...  
vector<int>::iterator itr = v.begin();  
++itr;  
int val = *itr;  
--itr;  
int val2 = *itr;
```

Use cases:

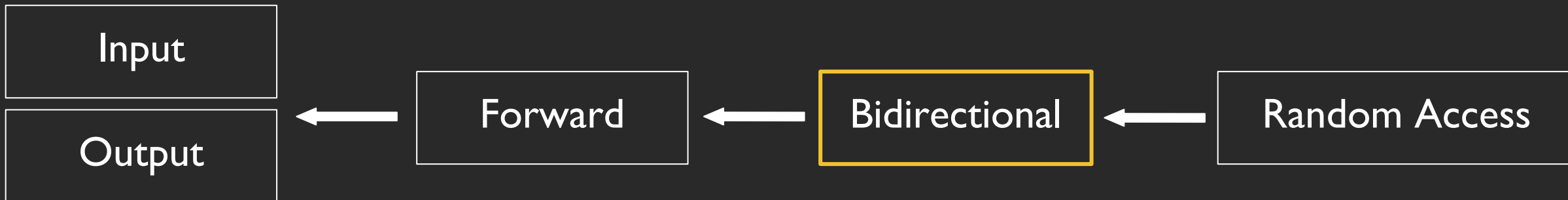
- reverse:

```
template< class BidirIt >  
void reverse( BidirIt first, BidirIt last );
```
- std::map, std::set
- std::list (sequence container, think of as doubly-linked list)

Bidirectional Iterators

Same as forward iterators, + can go backwards with the decrement operator (`--`) .

```
vector<int> v = ...  
vector<int>::iterator itr = v.begin();  
++itr;  
int val = *itr;  
--itr;  
int val2 = *itr;
```

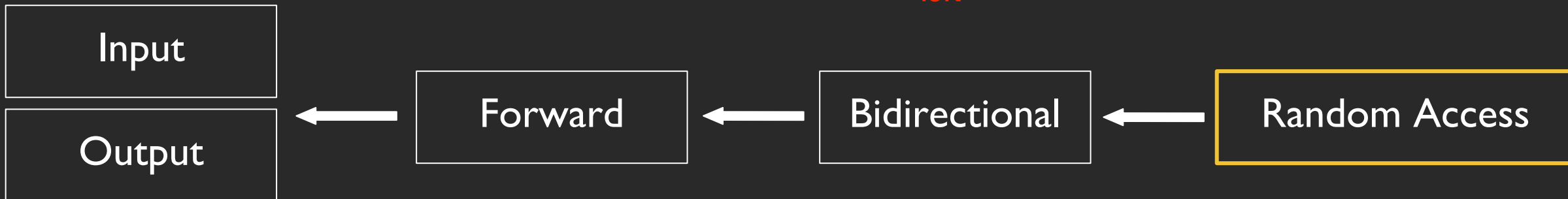


Random Access Iterators

Same as bidirectional iterators, + can be incremented or decremented by **arbitrary** amounts using + and -.

```
vector<int> v = ...  
vector<int>::iterator itr = v.begin();  
int val = *itr;  
itr = itr + 3;  
int val2 = *itr;
```

meaning of the arrow: the right is a sub-type of the left



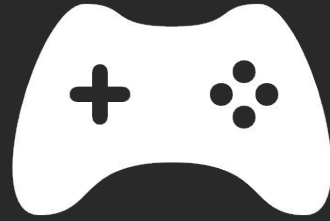
Random Access Iterators

Same as bidirectional iterators, + can be incremented or decremented by **arbitrary** amounts using + and -.

```
vector<int> v = ...  
vector<int>::iterator itr = v.begin();  
int val = *itr;  
itr = itr + 3;  
int val2 = *itr;
```

Use cases:

- std::vector, std::deque, std::string
- Pointers!



Next time

Templates!