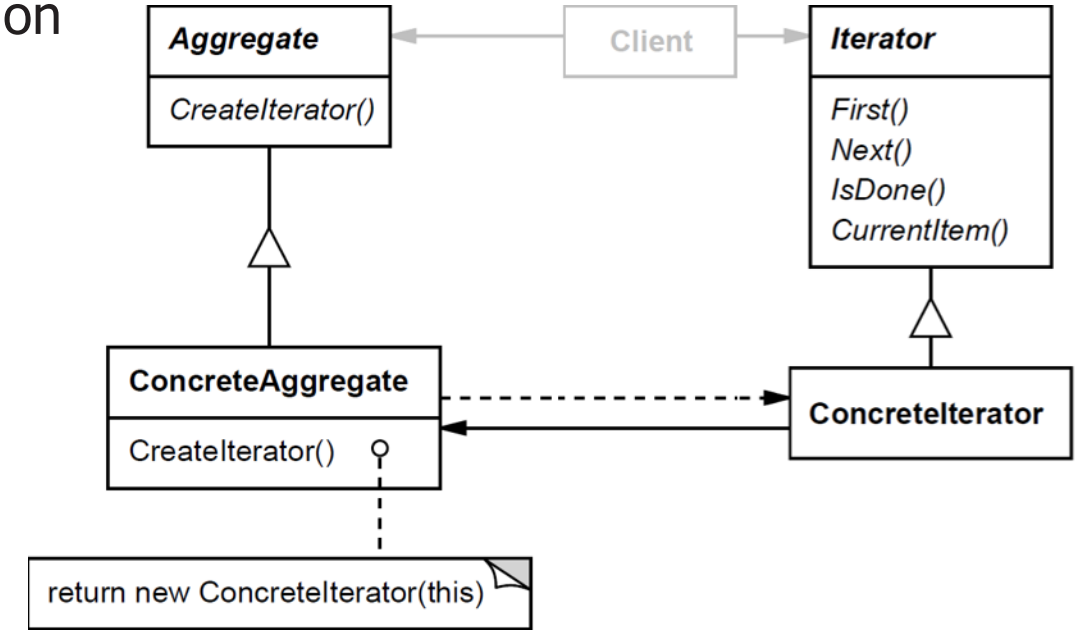
STL Iterator Overview

STL Iterator Overview

- STL iterators are a C++ implementation of the *Iterator pattern*



See en.wikipedia.org/wiki/Iterator_pattern

STL Iterator Overview

- STL iterators are a C++ implementation of the *Iterator pattern*
  - This pattern provides access to the elements of an aggregate object sequentially without exposing its underlying representation

```
vector<int> v{1, 2, 3, 4, 5};

for (vector<int>::iterator itr =
       v.begin();
     itr != v.end();
     ++itr)
  cout << *itr;
```

STL Iterator Overview

- STL iterators are a C++ implementation of the *Iterator pattern*
  - This pattern provides access to the elements of an aggregate object sequentially without exposing its underlying representation
  - An Iterator object encapsulates the internal structure of how the iteration occurs

```
template<typename T, ...>
class vector {
  class iterator {
  public:
    using reference  = T&;
    iterator(vector<T>* v, int i)
      : v_(v), i_(i) {}
    reference operator*()
    { return (*v_)[i_]; }
    iterator& operator++()
    { ++i_;return *this; }
    ...
  private:
    vector<T>* v_;
    int        i_;
  }; ...
```

See codereview.stackexchange.com/a/202165

STL Iterator Overview

- STL iterators are a generalization of pointers
  - i.e., they are objects that "point to" other objects

```
template<typename T, ...>
class vector {
  class iterator {
  public:
    using reference  = T&;
    iterator(vector<T>* v, int i)
      : v_(v), i_(i) {}
    reference operator*()
    { return (*v_)[i_]; }
    iterator& operator++()
    { ++i_;return *this; }
    ...
  private:
    vector<T>* v_;
    int        i_;
  }; ...
```

STL Iterator Overview

- Iterators are often used to iterate over a range of objects

```
template<typename InputIterator,
         typename OutputIterator>
OutputIterator copy
  (InputIterator first,
   InputIterator last,
   OutputIterator result) {
  for (; first != last;
       ++first, ++result)
    *result = *first;
    return result;
}
```

STL Iterator Overview

- Iterators are often used to iterate over a range of objects
  - i.e., if an iterator points to one element in a range it is possible to increment it so that it points to the next element

```
template<typename InputIterator,
         typename OutputIterator>
OutputIterator copy
  (InputIterator first,
   InputIterator last,
   OutputIterator result) {
  for (; first != last;
       ++first, ++result)
    *result = *first;
    return result;
}
```

STL Iterator Overview

- Iterators are central to generic programming because they are an interface between containers & algorithms

```
template<typename InputIterator,
         typename OutputIterator>
OutputIterator copy
  (InputIterator first,
   InputIterator last,
   OutputIterator result) {
  for (; first != last;
       ++first, ++result)
    *result = *first;
    return result;
}

vector<int> v1 {1, 2, 3, 4, 5, 6};
vector<int> v2 (v1.size());
copy(v1.begin(), v1.end(),
     v2.begin());
```

STL Iterator Overview

- Iterators are central to generic programming because they are an interface between containers & algorithms
  - Algorithms typically take iterators as arguments, so a container need only provide a way to access its elements using iterators

```
template<typename InputIterator,
         typename OutputIterator>
OutputIterator copy
  (InputIterator first,
   InputIterator last,
   OutputIterator result) {
  for (; first != last;
       ++first, ++result)
    *result = *first;
    return result;
}

vector<int> v1 {1, 2, 3, 4, 5, 6};
vector<int> v2 (v1.size());
copy(v1.begin(), v1.end(),
     v2.begin());
```

STL Iterator Overview

- Iterators are central to generic programming because they are an interface between containers & algorithms
  - Algorithms typically take iterators as arguments, so a container need only provide a way to access its elements using iterators
  - This enables generic algorithms that operate on many types of containers
    - e.g., a vector, list, deque, etc.

```
template<typename InputIterator,
         typename OutputIterator>
OutputIterator copy
  (InputIterator first,
   InputIterator last,
   OutputIterator result) {
  for (; first != last;
       ++first, ++result)
    *result = *first;
    return result;
}

vector<int> v {1, 2, 3, 4, 5, 6};
list<int> l (v.size());
copy(v.begin(), v.end(),
     l.begin());
```

STL Iterator Categories

- Iterator *categories* depend on C++ type parameterization instead of inheritance

Iterator categories	Provider
Input iterator	istream
Output iterator	ostream
Forward iterator	
Bidirectional iterator	list, set, multiset, map, multimap
Random access iterator	vector, deque, array

STL Iterator Categories

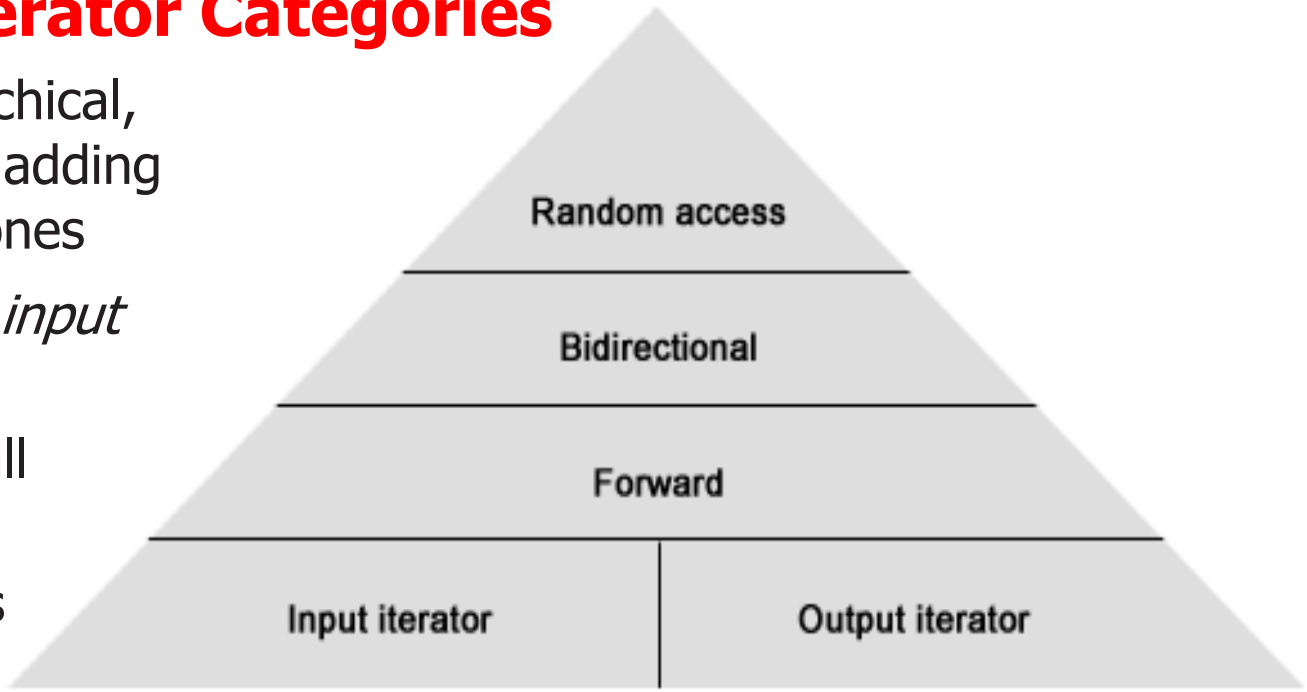
- Iterator *categories* depend on C++ type parameterization instead of inheritance
  - Algorithms can thus operate seamlessly on both native (i.e., pointers) & user-/STL-defined iterator types

```
template<typename InputIterator,
         typename OutputIterator>
OutputIterator copy
  (InputIterator first,
   InputIterator last,
   OutputIterator result) {
  for (; first != last;
       ++first, ++result)
    *result = *first;
    return result;
}

int a[] = {1, 2, 3, 4, 5, 6};
list<int> l (end(a) - begin(a));
copy(begin(a), end(a),
     l.begin());
```

STL Iterator Categories

- Iterator categories are hierarchical, with more refined categories adding constraints to more general ones
  - *Forward* iterators are both *input* & *output* iterators
  - *Bidirectional* iterators are all *forward* iterators
  - All *random-access* iterators are *bidirectional* iterators



Simple STL Iterator Example

```
#include <iostream>
#include <vector>
#include <string>
int main (int argc, char *argv[]) {
  std::vector <std::string> projects;// Names of the projects

  for (int i = 1; i < argc; ++i)
      projects.emplace_back (std::string (argv [i]));

  for (std::vector<std::string>::iterator j = projects.begin ();
       j != projects.end (); ++j)
      std::cout << *j << std::endl;

  return 0;
}
```

See github.com/douglascraigschmidt/CPlusPlus/tree/master/STL/S-04/4.2