STL Random-Access Iterators
STL Random-Access Iterators

- *Random-access* iterators allow algorithms random-access to elements stored in a container that provides *random-access* iterators
  
  - *e.g., vector* & *deque*

```cpp
vector<string> v {"a", "b", "c", "d", ...
... "j"};

auto p = v2.begin();
p += 5
p[1] = "z";
p -= 5;
```

STL Random-Access Iterators

- Random-access iterators must implement the requirements for bidirectional iterators, plus:
  - Arithmetic assignment operators `+=` and `-=`
  - Operators `+` and `-` (must handle symmetry of arguments)
  - Relational operators `<` and `>`, `&` and `<=` and `>=`
  - Subscript operator `[ ]`
STL Random-Access Iterator Example

```cpp
int main() {
    vector<int> v {1, 2, 3, 4};
    auto i = v.begin();
    auto j = i + 2; cout << *j << " ";
    i += 3; cout << *i << " ";
    j = i - 1; cout << *j << " ";
    j -= 2;
    cout << *j << " " << v[1] << endl;
    j < i ? cout << "j < i" : cout << "not (j < i)"; cout << endl;
    j > i ? cout << "j > i" : cout << "not (j > i)"; cout << endl;
    i = j;
    i <= j && j <= i ? cout << "i and j equal" : cout << "i and j not equal";
    cout << endl;
}```

See github.com/douglas craigschmidt/CPlusPlus/tree/master/STL/S-04/4.7
Implementing Iterators Using STL Patterns

- A C++ iterator provides a familiar & standard interface, so you may want to add one to your own classes so you can “plug-and-play” with STL algorithms

```cpp
template <typename T>
class ArrayListIterator : public std::iterator<
   std::random_access_iterator_tag, T> {
public:
   T &operator*();
   const T &operator*() const;
   T *operator->();
   const T *operator->() const;
   ArrayListIterator<T> &operator++();
   ArrayListIterator<T> &operator--();
   ...
private:
   explicit ArrayListIterator(T *ptr);
   T *mPtr;
};
```

See users.cs.northwestern.edu/~riesbeck/programming/c++/stl-iterator-define.html
Implementing Iterators Using STL Patterns

• Writing your own iterators is a straight-forward (albeit tedious) process, with only a few subtleties you should be aware of

• e.g., which category to support, pre- & post-increment/decrement operators, etc.

```cpp
template <typename T>
class ArrayListIterator : public std::iterator<std::random_access_iterator_tag, T> {
    public:
    T &operator*();
    const T &operator*() const;
    T *operator->();
    const T *operator->() const;
    ArrayListIterator<T> &operator++();
    ArrayListIterator<T> &operator--();
    
    private:
    explicit ArrayListIterator(T *ptr);
    T *mPtr;
};
```