Standard Template Library (STL)

Introducing STL Containers

- Predefined templates that can store virtually any type of data
- The appropriate container will be dictated by the application requirements
- Example considerations:
  - Does the data need to be sorted?
  - How will the data be accessed?
    - Front to back?
    - Randomly?
  - Will additional data ever need to be added or removed?
- Careful planning will allow you to write clean, efficient code

Types of Containers

- Sequence containers
  - Programmer controls the order of the elements
- Associative containers
  - Position of elements is controlled by container
  - Elements are generally accessed by using a “key”
- Technical detail…
  - There are also container “adaptors”. This is when you use an existing type of container to build other types.
  - In this context, we call these “Abstract Data Types”

Examples of C++ Containers

- Some common containers:
  - <array> - Stores a constant amount of data in contiguous memory
  - <vector> - An array that can be resized
  - <list> - Linked list that stores data in non-contiguous memory
  - <set> - An ordered collection of items
  - <queue> - Stores data & returns it in the order it was received
  - <stack> - Returns data in the opposite order that it was received in
- Generally a good idea to refer to the Standard Template Library (STL) documentation before starting a project

Examples of C++ Containers

- Some common containers:
  - <array> - Stores a constant amount of data in contiguous memory
  - <vector> - An array that can be resized
  - <list> - Linked list that stores data in non-contiguous memory
  - <set> - An ordered collection of items
  - **<queue> - Stores data & returns it in the order it was received**
  - <stack> - Returns data in the opposite order that it was received in
- Generally a good idea to refer to the Standard Template Library (STL) documentation before starting a project

Examples of C++ Containers

- Some common containers:
  - <array> - Stores a constant amount of data in contiguous memory
  - <vector> - An array that can be resized
  - <list> - Linked list that stores data in non-contiguous memory
  - <set> - An ordered collection of items
  - <queue> - Stores data & returns it in the order it was received
  - **<stack> - Returns data in the opposite order that it was received in**
- Generally a good idea to refer to the Standard Template Library (STL) documentation before starting a project

Basic Iteration Techniques

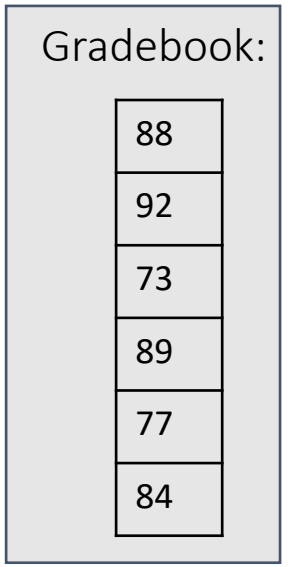
- Traditional loops can be used to access containers
  - *for* loop
  - *for each* loop
- *for* loops often provide the most concise code
  - Variables can be instantiated and incremented in a single line
- Will demonstrate some examples using a fictional gradebook

Basic Iteration Techniques – *for* loop

```
array<int, 6> gradebook = {88, 92, 73, 89, 77, 84};

cout << "Gradebook:" << endl;
for (int i = 0; i < gradebook.size(); i++) {
  cout << gradebook[i] << endl;
}
```

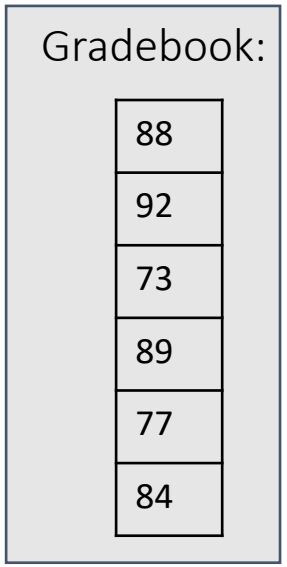
This approach is concise and helps to clearly communicate your intention.



Basic Iteration Techniques – *for-each* loop

```
array<int, 6> gradebook = {88, 92, 73, 89, 77, 84};

cout << "Gradebook:" << endl;
for (int grade : gradebook) {
  cout << grade << endl;
}
```



A *for-each* loop is a useful technique when you want to access **each** element in a container.

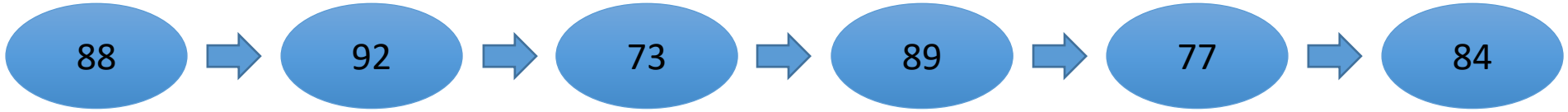
Note:
for-each loops were added in C++11 so old compilers will not work.

A New Approach: Introducing Iterators

- As the name implies, iterators allow you to iterate through elements
  - Pointers are a specific type of iterator
- There are several common types of iterators:
  - ***Forward Iterators*** can only iterate forward
  - ***Bidirectional Iterators*** can iterate in either direction
  - ***Random Access Iterators*** can do all of the above (and more)!
- C++ allows you to easily create iterators for any container
  - Implementation details may limit the types of iterators that can be used
  - STL documentation shows which specific iterator features are supported

Why are iterators useful?

- Iterators are an abstraction to hide complexity
  - Makes code easier to read
  - Can reduce the potential for bugs
- Some data structures can’t support random access
  - Iterators are necessary to access the inner elements
  - Consider a forward linked list:



Accessing an Array – Iterator Technique

```
array<int, 6> gradebook = {88, 92, 73, 89, 77, 84};

cout << "Gradebook:" << endl;
for (auto it = gradebook.begin(); it != gradebook.end(); it++) {
  cout << *it << endl;
}
```

In this instance the "auto" keyword is equivalent to writing "array<int, 6>::iterator"

Gradebook:

88
92
73
89
77
84

Note:
This code could be used to iterate through any container that supports **Forward Iterators**.

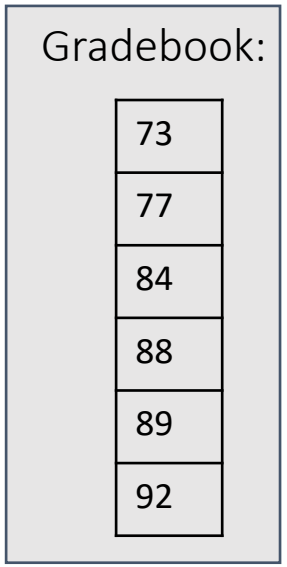
Applying our New Knowledge

- Suppose that we want to display a sorted printout of student grades
  - Assume that grades should be displayed in ascending order
- The C++ ***multiset*** container is well suited for this task
  - It automatically sorts the values as they are added
- Our code can simply iterate over the container and display each grade
  - The *for-each* technique will automatically use iterators for us

Displaying a Sorted Set of Grades

```
multiset<int> gradebookSet;
gradebookSet.insert(88);
gradebookSet.insert(92);
gradebookSet.insert(73);
gradebookSet.insert(89);
gradebookSet.insert(77);
gradebookSet.insert(84);

cout << "Gradebook:" << endl;
for (int grade : gradebookSet) {
  cout << grade << endl;
}
```



Using Reverse Iterators

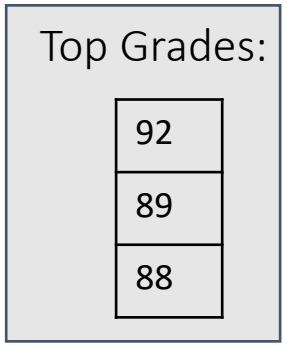
- As noted earlier, bidirectional iterators can work in either direction
- This functionality allows us to easily access elements in reverse
  - "Incrementing" a reverse iterator moves in reverse
- Suppose that we wish to display only the 3 top grades
  - The *for-each* technique will not work
  - This is very simple if we use the reverse iterator rbegin()

`rbegin()`  `begin()`

Displaying the Top Three Grades

```
multiset<int> gradebookSet;
gradebookSet.insert(88);
gradebookSet.insert(92);
gradebookSet.insert(73);
gradebookSet.insert(89);
gradebookSet.insert(77);
gradebookSet.insert(84);

cout << "Top Grades:" << endl;
auto it = gradebookSet.rbegin();
for (int i = 0; i < 3; i++) {
  cout << *it << endl;
  it++;
}
```



Summary of Iterators

- Iterators offer a powerful way to access data within containers
  - In some cases, iteration is the only option
- The abstract nature of iterators simplifies code
  - The underlying data structure does not matter
  - Resulting code is easier to proofread and comprehend
- As a programmer you should strive to write clean, concise code
  - Iterators are one of the tools that can help you