Stack and Queue

- This topic teaches how to use perform the following with Stacks and Queues
  - Push, pop, push_back, pop_back, empty, size, top
- It also teaches how to use a priority_queue

- container adapters
  - `Stack`, `queue` and `priority_queue`
  - *Not first-class containers*
    - Do not provide the actual data-structure implementation in which elements can be stored
    - Adapters do *not* support iterators
- The benefit of an *adapter class*
  - You can choose an appropriate underlying data structure
- All three *adapter classes* provide member functions `push` and `pop`
  - Insert an element into each adapter data structure and remove an element from each adapter data structure

- Class `stack` (from header `<stack>`)
  - Enables insertions into and deletions from the underlying container at one end called the *top*
  - *last-in, first-out* (LIFO)
  - Can be implemented with `vector`, `list` or `deque`
- By default, a `stack` is implemented with a `deque`

- The `stack` operations are
  - `push` to insert an element at the *top* of the `stack`
    - Calls `push_back` of the underlying container
  - `pop` to remove the top element of the `stack`
    - Calls `pop_back` of the underlying container
  - `top` to get a reference to the top element of the `stack`
    - Calls `back` of the underlying container
  - `empty` to determine whether the `stack` is empty
    - Calls `empty` of the underlying container
  - `size` to get the number of elements in the `stack`
    - Calls `size` of the underlying container

- Figure 15.19 demonstrates the `stack` adapter class.

- A queue is similar to a *waiting line*
  - The item that has been in the queue the *longest* is the *next* one removed—so a queue is referred to as a first-in, first-out (FIFO) data structure
- Class `queue` (from header `<queue>`)
  - Enables insertions at the *back* of the underlying data structure and deletions from the *front*
- A `queue` can store its elements in objects of the Standard Library’s `list` or `deque` containers
- By default, a `queue` is implemented with a `deque`

- The common `queue` operations are
  - `push` to insert an element at the back of the `queue`
    - Calls `push_back` of the underlying container
  - `pop` to remove the element at the front of the `queue`
    - Calls `pop_front` of the underlying container
  - `front` to get a reference to the first element in the `queue`
    - Calls `front` of the underlying container
  - `back` to get a reference to the last element in the `queue`
    - Calls `back` of the underlying container
  - `empty` to determine whether the `queue` is empty
    - Calls `empty` of the underlying container
  - `size` to get the number of elements in the `queue`
    - Calls `size` of the underlying container

- Figure 15.20 demonstrates the `queue` adapter class.

- Class `priority_queue` (from header `<queue>`)
  - Provides functionality that enables
    - *insertions* in *sorted order* into the underlying data structure
    - deletions from the *front* of the underlying data structure
- By default, a `priority_queue`'s elements are stored in a `vector`
- When elements are added to a `priority_queue`
  - They're inserted in *priority order*
  - The highest-priority element (i.e., the *largest* value) will be the first element removed

- Priority order is accomplished by arranging the elements in a data structure called a heap
  - Not to be confused with the heap for dynamically allocated memory
  - Always maintains the largest value (i.e., highest-priority element) at the front of the data structure

- The comparison of elements is performed with *comparator function object* `less< T >` by default
  - But you can supply a different comparator

- There are several common `priority_queue` operations
  - Function `push` inserts an element at the appropriate location based on *priority order* of the `priority_queue`
    - Calls `push_back` of the underlying container, which then reorders the elements in priority order
  - `pop` removes the *highest-priority* element of the `priority_queue`
    - `Calls pop_back` of the underlying container after removing the top element of the heap
  - `top` gets a reference to the *top* element of the `priority_queue`
    - `Calls front` of the underlying container
  - `empty` determines whether the `priority_queue` is *empty*
    - Calls `empty` of the underlying container
  - `size` gets the number of elements in the `priority_queue`
    - `Calls size` of the underlying container

- Figure 15.21 demonstrates the `priority_queue` adapter class
- Header `<queue>` must be included to use class `priority_queue`

- This topic taught and demonstrated how to perform the following with Stacks and Queues
  - Push, pop, push_back, pop_back, empty, size, top
- It also taught and demonstrated how to use a priority_queue