Stacks And Queues, Reading

Stacks and queues are much like the arrays and the templates we studied in earlier modules. But the are adapted to very specific applications. Like arrays, they store values that are all of the same data type. But unlike arrays, they are **not meant to allow access to any values except the oldest (queue) or the newest (stack)**. When values get added, we really don't care where they are stored or how they are stored, for that matter. We just want to put values in, be able to check how many values there are, and retrieve the oldest (or newest), period.

Stacks (LIFO)

Stacks are a **"last in, first out"** data structure (LIFO). Whatever was the last value stored, that's the one to be retrieved and/or removed. It is meant to model real-life "stacks" of... stuff. A good example is a discard pile in a game with playing cards. In a stack of playing cards, only the top card is visible and accessible. Only the top card can be removed. Additions to the pile are made at the top. So if a playing card is added to the stack, *it's* now the newest one in the stack, and it's the one that's visible and can possibly be removed.

In card games, players normally **do not look through the stack** -- again, only the top card is visible.

Yes, we can use arrays to do the same thing. But stack data structures are specially designed to make the coding easier.

Here's a **specification**, using a PlayingCard object to demonstrate:

```
PlayingCard a = {"Ace", "Spade"};
PlayingCard b = {"three", "Club"};
PlayingCard c = {"Queen", "Diamond"};
PlayingCard d = {"ten", "Spade"};

Stack<PlayingCard> discardPile;
discardPile.push(a); // copies ace to the stack
discardPile.push(b); // copies 3 on top of the ace
discardPile.push(c); // copies queen on top of 3
discardPile.push(d); // top card is the 10 of spades

PlayingCard& topCard =  discardPile.peek( ); // alias for top card, the 10 of spades
PlayingCard topCardCopy = discardPile.peek( ); // a copy of top card, still the 10 of spades
discardPile.pop( ); // loses the 10 of spades, top is now the queen
cout << discardPile.size( ); // 3 cards are left
if (discardPile.empty( )) ... // checks if size is zero
discardPile.clear( ); // loses all cards
```

If you've not heard of them before, the terms **"push", "peek", and "pop"** seem like odd choices for function names. But they are common terms associated with stacks (ref. **wikipedia.org (https://en.wikipedia.org/wiki/Stack_(abstract_data_type))**). And look what's missing -- there's *no square bracket operator*. There's no way provided for looking at anything other than the last-added value.

Those three operations are standard for a stack, although every implementation may not use these exact words. In fact, the C++ STL implementation uses the more descriptive **"top" instead of "peek"**. Also the implementation of these operations can vary. For example, the "pop" as shown above and as implemented in the STL's stack is a simple void function. It returns nothing. It just discards the top value in the stack without looking at it or returning it or anything. If the programmer wants to "see" the value before popping it, they have to peek at it first. But other implementations return a copy either in a return statement or as a mutable (that is, not const) reference parameter.

Other non-standard operations are often included. The STL's implementation has "**empty**" and "**size**" as getters. Our implementation does too, plus ours has a "**clear**" setter to pop *all* the values at once.

Queues (FIFO)

Queues are a **"*first* in, first out"** data structure (FIFO). Whatever was the *first* value stored, that's the one to be retrieved and/or removed. It is meant to model real-life "waiting queues". A good example is a line at a movie theater where people are in line to buy tickets. The first arriving customer is at the front of the line, waiting to be the next to be served (no cuts!).

To the "server" -- the one processing the values in the queue -- **only the first "customer" is visible**. It might be nice to know how may are in the queue, and it might be nice to disperse the queue in one action (like closing the service window in the server's booth and leaving), but there's no need to *access* any other than the first in line. So no operator square bracket here either.

Here's a **specification**, using a Customer object to demonstrate:

```
Customer a = {"Joe", "10:15"}; // name and arrival time
Customer b = {"Jane", "10:16"};
Customer c = {"Fred", "10:21"};
Customer d = {"Jill", "10:22"};

Queue<Customer> waitingLine;
waitingLine.push(a); // copies Joe to the queue
waitingLine.push(b); // adds Jane behind Joe
waitingLine.push(c); // adds Fred behind Jane
waitingLine.push(d); // adds Jill behind Fred

Customer& firstInLine =  waitingLine.front( ); // alias for Joe
Customer& lastArrival =  waitingLine.back( ); // alias for Jill
waitingLine.pop( ); // done with Joe -- Jane's next
cout << waitingLine.size( ); // 3 customers are left
if (waitingLine.empty( )) // ...checks if size is zero
waitingLine.clear( ); // sends all remaining customers home
```

The specification is a lot like the stack's, but "peek" is replaced by **"front" and "back"**. The latter is not really needed -- the server cannot see anybody but the *first* customer. But "back" is often included, and the STL's queue has it.