#pragma once

template <typename T>
class List {
  public:
    /* ... */
  private:
};
Array Implementation

C S 2 2 5
[0] [1] [2] [3] [4]
Array Implementation

**insertAtFront:**

<table>
<thead>
<tr>
<th>C</th>
<th>S</th>
<th>2</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
</tr>
</tbody>
</table>
Resize Strategy: +2 elements every time
Resize Strategy: +2 elements every time
Resize Strategy: x2 elements every time
Resize Strategy: x2 elements every time
<table>
<thead>
<tr>
<th>Array Implementation</th>
<th>Singly Linked List</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert/Remove at <strong>front</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert at <strong>given</strong> element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove at <strong>given</strong> element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert at <strong>arbitrary</strong> location</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove at <strong>arbitrary</strong> location</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
std::vector
**std::vector**

Defined in header `<vector>`

```cpp
template<
    class T,
    class Allocator = std::allocator<T>
>
> class vector;
```

```cpp
namespace pmr {
    template <class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```

1) `std::vector` is a sequence container that encapsulates dynamic size arrays.

2) `std::pmr::vector` is an alias template that uses a polymorphic allocator

The elements are stored contiguously, which means that elements can be accessed not only through iterators, but also using offsets to regular pointers to elements. This means that a pointer to an element of a vector may be passed to any function that expects a pointer to an element of an array.

The storage of the vector is handled automatically, being expanded and contracted as needed. Vectors usually occupy more space than static arrays, because more memory is allocated to handle future growth. This way a vector does not need to reallocate each time an element is inserted, but only when the additional memory is exhausted. The total amount of allocated memory can be queried using `capacity()` function. Extra memory can be returned to the system via a call to `shrink_to_fit()` (since C++11)
# Element access

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>at</code></td>
<td>access specified element with bounds checking</td>
</tr>
<tr>
<td><code>operator[]</code></td>
<td>access specified element</td>
</tr>
<tr>
<td><code>front</code></td>
<td>access the first element</td>
</tr>
<tr>
<td><code>back</code></td>
<td>access the last element</td>
</tr>
<tr>
<td><code>data(C++11)</code></td>
<td>direct access to the underlying array</td>
</tr>
</tbody>
</table>

# Capacity

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>empty</code></td>
<td>checks whether the container is empty</td>
</tr>
<tr>
<td><code>size</code></td>
<td>returns the number of elements</td>
</tr>
<tr>
<td><code>max_size</code></td>
<td>returns the maximum possible number of elements</td>
</tr>
<tr>
<td><code>reserve</code></td>
<td>reserves storage</td>
</tr>
<tr>
<td><code>capacity</code></td>
<td>returns the number of elements that can be held in currently allocated storage</td>
</tr>
<tr>
<td><code>shrink_to_fit(C++11)</code></td>
<td>reduces memory usage by freeing unused memory</td>
</tr>
</tbody>
</table>

# Modifiers

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clear</code></td>
<td>clears the contents</td>
</tr>
<tr>
<td><code>insert</code></td>
<td>inserts elements</td>
</tr>
<tr>
<td><code>emplace(C++11)</code></td>
<td>constructs element in-place</td>
</tr>
<tr>
<td><code>erase</code></td>
<td>erases elements</td>
</tr>
<tr>
<td><code>push_back</code></td>
<td>adds an element to the end</td>
</tr>
<tr>
<td><code>emplace_back(C++11)</code></td>
<td>constructs an element in-place at the end</td>
</tr>
<tr>
<td><code>pop_back</code></td>
<td>removes the last element</td>
</tr>
<tr>
<td><code>resize</code></td>
<td>changes the number of elements stored</td>
</tr>
<tr>
<td><code>swap</code></td>
<td>swaps the contents</td>
</tr>
</tbody>
</table>
Stack ADT
Queue ADT
#pragma once

#include <vector>

template <typename T>
class Stack {
   public:
      void push(T & t);
      T & pop();
      bool isEmpty();

   private:
      std::vector<T> list_;}

#include "Stack.hpp"
Stack Implementation

```cpp
3 template <typename T>
4 void Stack<T>::push(const T & t) {
5     list_.push_back(t);
6 }

8 template <typename T>
9 const T & Stack<T>::pop() {
10    const T & data = list_.back();
11    list_.pop_back();
12    return data;
13 }
```
Implications of Design

1. ```cpp
class ListNode {
    public:
    T & data;
    ListNode * next;
    ...
}
```

2. ```cpp
class ListNode {
    public:
    T * data;  ...
}
```

3. ```cpp
class ListNode {
    public:
    T data;  ...
}
```
# Implications of Design

<table>
<thead>
<tr>
<th></th>
<th>Storage by Reference</th>
<th>Storage by Pointer</th>
<th>Storage by Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Who manages the lifecycle of the data?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is it possible for the data structure to store NULL?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If the data is manipulated by user code while in our data structure, is the change reflected in our data structure?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is it possible to store literals?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speed</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Lifecycle

Storage by reference:

```
1 Sphere s;
2 myStack.push(s);
```

Storage by pointer:

```
1 Sphere s;
2 myStack.push(&s);
```

Storage by value:

```
1 Sphere s;
2 myStack.push(s);
```
Possible to store NULL?

Storage by reference:

```cpp
class ListNode {
    public:
        T & data;
        ListNode * next;
    ListNode(T & data) : data(data), next(NULL) { }
};
```

Storage by pointer:

```cpp
T ** arr;
```

Storage by value:

```cpp
T * arr;
```
Data Modifications

```cpp
1  Sphere s(1);
2  myStack.push(s);
3  
4  s.setRadius(42);
5  
6  Sphere r = myStack.pop();
7  // What is r’s radius?
```
Speed