CSCI 152, Performance and Data Structures, Assignment 2

Rules for Assignments:

- You are expected to read the complete assignment and to complete every part of it. 'I did not see this part' is never a valid excuse for not completing a part of the assignment.
- Submitted code will be checked for correctness, correct memory management, readability, style, and layout. We use **valgrind** for memory checks. We apply high quality standards during grading.
- Assignments are graded with the help of autograders. Make sure that your submitted code can be compiled. If you don't know how to complete a task, insert an empty function, so that your code compiles in all cases.
- In order to make it possible to adopt your code to our automatic checkers, and to give more feedback, we check your code at least twice. The last submitted version counts. Use the earlier checks. If you submit only for the last check, you are taking unnecessary risk.
- If you need help, first your question on Piazza. Questions that do not contain code can be asked publicly. Questions that contain code must be posted as private question. Don't mail questions directly to an instructor, because individual instructors are not always available. Piazza is monitored almost full time, so you will get a quicker answer.
- Don't wait until the last moment with starting to work on the assignment.
- You must write all submitted code by yourself! Even allowing others to see your code, or looking at others' code is already academic misconduct.

C^{++} Coding Rules

- Avoid unitialized variables. If you don't have a value to initialize a variable, you are probably declaring it too early. C^{++} allows declarations of variables almost everywhere.
- Make sure to use `size_t` for all indexing. (Not `int`, nor `unsigned int`.)
- Do not implement anything in header files. In real life, short methods can be implemented in header files, while longer methods should be implemented in **.cpp** files. We test with our own header files, so everything that you write in a header file, will be ignored.
- You are not allowed to use any STL containers or library functions, unless explicitly mentioned in the assignment. If in doubt, ask in Piazza.
- Don't use `printf`, `malloc`, `free`, or `memcpy`. Don't use `#define`, except in include guards.
- Don't use `0` or `NULL` to represent the null-pointer. The only correct representation of the null-pointer is `nullptr`. It is OK to write `if(p)` if you want to test that `p` is not the null-pointer.
- Avoid `break` and `continue`. They are just `goto` in disguise. In nearly all cases, they can be avoided by changing the condition of the **while** loop, which always results in more readable code.
- Don't `throw` and `catch` an exception inside the same function. That is not what exceptions are for.
- Avoid assignments in constructors. Use member initializers wherever possible.
- Don't write: `if(b) return true; else false;` Just write `return b;`

1 Introduction

In this assignment you will implement a *double ended queue* (deque) by means of a *doubly linked list.* This means that deque is the ADT, and doubly linked list is the implementation method. You will also become familiar with an important feature of C^{++}, namely *moving.*

As with the previous assignment, we want to implement the ADT as generic as possible, so we will use a type variable `valtype`, which can be assigned with different concrete types. It should be tested as least with `valtype = double`, and `valtype = order`. This will happen in part 3.

A double ended queue combines stack and queue into a single data structure. Values can be pushed and popped from both ends of the deque. In order to distinguish the sides, `push` will be renamed into `push_front` and `push_back`. Method `peek` will be split into `front( )` and `back( )`. In order to be more consistent with the STL, both functions will have two versions, **const** and **non-const**. Method `pop( )` will be split into `pop_front( )` and `pop_back( )`.

As usual, the implementation of deque must have value semantics, so that you will need to write the copy constructor and assignment operator, in addition to the specific deque methods.

You will also implement the moving copy constructor and assignment. This will happen in Part 4.
Download the files **order.h**, **price.h**, **deque.h**, **deque.cpp**, and **main.cpp**. Also download the **Makefile**.

Take a look at file **deque.h** and make sure that you understand it. `struct node` is defined inside `class deque`, as a private member, so that the user of `deque` cannot access it directly.

In order to make testing from main possible, we declared `main( )` as friend. Purpose of `noexcept` is to promise that a function will never throw an exception. This qualifier should be added to constructors and assignment operators that cannot fail, and do not allocate heap memory.

2 Tasks

We give the tasks in the easiest order, so we recommend that you follow our order when making the tasks.

1. Implement the default constructor `deque( )`. It should construct an empty deque.

2. Implement `void push_back( const valtype& val )`. Allocate a new node with `n` inside, check `deq_size`, and adjust the pointers.

3. Implement `void print( std::ostream& out ) const`. The output must look as follows:

```
deque q;
```

```
std::cout << q << "\n";
   // Prints  [ ]
q. push_back(1);
std::cout << q << "\n";
   // Prints [ 1 ]
q. push_back(2);
std::cout << q << "\n";
   // Prints [ 1, 2 ].
q. push_front(3);
std::cout << q << "\n";
   // Prints [ 3, 1, 2 ].
```

4. Implement `pop_back( )`. If you want, you can throw an exception when `deq_size == 0`.

   After that, distinguish between `deq_size == 1` and `deq_size > 1`. We recommend that you use `valtype std::exchange( valtype& v1, const valtype& v2 )`, which assigns `v1 = v2`, and returns the old value of `v1`.

5. Implement `const valtype& back( ) const`.

   If you want, you may throw an exception if the deque is empty.

6. Also implement the non-const version `valtype& back( ) const`. Test that you can assign to `back`.

   Test what you have until now very carefully, use method `check_invariant( ) const`.

7. Now you can implement all of

   ```
   push_front( const valtype& val );
   pop_front( );
   valtype& front( );
   const valtype& front( ) const;
   ```

   This can be easily done by making copies of all the back methods, and taking the mirror image. This means exchanging `front` and `back` everywhere, and exchanging `next` and `prev`.

8. It's time to implement the destructor `~deque( )`. Implementation can be simplified using `std::exchange`, but it can also be written without.

9. Implement the copy constructor `deque( const deque& q )`. The easiest way is to construct an empty deque, and push the elements of `q` to the back.

10. Implement the initializer list constructor `deque( std::initializer_list< valtype > init )`. Again, the easiest way is to construct an empty deque, and then push the elements of `q`.

11. Implement methods `reset_front( size_t s )` and `reset_back( size_t s )`. They pop elements from the front (or back) until the deque has the desired size.

12. Implement `operator = ( const deque& q )`. If you want to do it well, you can try to reuse as much of `*this` as possible. If you don't care about efficiency, you can just check for self assignment, delete all of `*this`, and push the values in `q`.

13. Implement `size( ) const`.

14. Implement `bool empty( ) const`. Don't write code of form `if(b) return true; else false;`

Test carefully, make sure that your tests include all deque methods, including copy constructor and assignment. Don't forget to check self assignment. Also test with `valgrind`.

3 Take a Break in the Coffee Bar

We need to test if it is possible to change `valtype`. Change the `#if` in main, and replace `valtype = double;` by `valtype = order`.

Type **order** consists of orders typical for a coffee bar. An order specifies the ordered product, the unit price for this product, and how many items were ordered. For example, one can order two cappuccino's, costing 850 tenge each. Unfortunately, this is a simplification, because in reality one can order different products in a single order, like a cappuccino and a croissant. We leave this for a future upgrade, so that for the moment, the client has to place two separate orders.

You don't need to test much, because the real testing was done with `valtype = double;` but you should check at least that your code compiles with `valtype = order;` During testing, we will likely use another `valtype`, and your code must work with every `valtype`.

4 Move Semantics

C^{++}-11 was the biggest extension in the history of C^{++}. Main additions were the initializer lists and initialization with `{ }`, the range for, the `auto` keyword, **constexpr** functions, and move semantics. You have seem most of these in action by now, but not yet the move semantics.

The main purpose of move semantics is to remove inefficiencies that are caused by value semantics, without losing value semantics.

Consider the implementation of **deque** that you wrote above. It consists of a small part on the stack (size with front and back pointer), and a part on the heap that is unbounded in size. There exist many data structures of this form, for example all STL containers have similar form, the **queue** that you wrote in assignment 1 has this form, and `std::string` has this form.

With this form in mind, consider the following situations:

- A statement `return q` that returns a deque from a function. The compiler will call the copy constructor of deque to copy the value of `q` to the context from where the function was called. After that, variable `q` will be destroyed.
- Resizing a `std::vector< deque >`. The vector will allocate a new heap space of type `deque*` , copy the existing deques to this new heap space, and destroy the old deques.
- Calling `std::exchange( q1, q2 )` with two deques. A possible implementation is

  ```
  template< typename T >
  T exchange( T& t1, const T& t2 )
  {
     auto old = t1;
     t1 = t2;
     return old;
  }
  ```

  The first statement `old = t1;` calls the copy constructor of deque with `t1` as argument. After that, `t1` is overwritten with the value of `t2`. Finally, `old` is returned, which calls the copy constructor of deque, using `old` as argument, after which `old` is destroyed.

All of these examples have the following problem:

The heap part of the data structure is copied (either by copy-constructor or assignment), and after that, the heap part is deleted (either by assigning a new value, or by the destructor). In all examples, copying could be easily avoided by passing the pointers into the new variable without copying the heap data. The efficiency improvement would be very big, because copying all heap data takes $O(n)$, while at the same time copying local data (the pointers plus some size info) takes $O(1)$. Just replacing the copy constructor by

```
deque( const deque& q )
   : deq_size( q. deq_size ),
     deq_front( q. deq_front ),
     deq_back( q. deq_back )
{ }
```

will not work, because it will create shared representation, which breaks value semantics.

We will call (passing the local data without copying heap data) *moving.* In C^{++}-11, language support for moving was added by creating a new type of reference, called *rvalue reference.* The syntax for **rvalue** reference is `T&&`, where `T` is the type being referred to. Unfortunately, there is no precise definition of what **rvalue** reference means, but you will get a long way with the following definition:

An **rvalue** reference is a reference to a variable that is about to be overwritten or destroyed. That means that the way in which the program will continue, does not depend on the value of the variable, as long as it fulfills the class invariant.

It follows from this definition that a function that receives an **rvalue** reference is allowed to change the value of the variable to any other value as long as it obeys the class invariant.

In practice, one should follow the following rule: A function that receives an **rvalue** reference should leave the variable in a state that holds as few as possible resources without breaking the class invariant.

Using **rvalue** reference, the following copy constructor can be defined:

```
deque( deque&& q )
   : deq_size( q. deq_size ),
     deq_front( q. deq_front ),
     deq_back( q. deq_back )
{
   q. deq_size = 0;
   q. deq_front = nullptr;
   q. deq_back = nullptr;
}
```

Note that a function that receives an **rvalue** reference is not allowed to destroy the object (This would be RUST, and it is problematic from the compiler point of view). The copy constructor above does not destroy `q`, it only makes `q` empty. There are three ways in which an **rvalue** reference can be obtained:

- When a variable is returned from a function, it is an **rvalue** reference.
- When the user writes `std::move(t)`, and `t` has type `T&`, the result will be of type `T&&`. If `t` has type `T` or `const T&`, `std::move` will not change the type.

  Unfortunately, the name `std::move` is misleading. It doesn't move by itself, it only changes the type into **rvalue** reference, which makes moving possible.
- Every subexpression that has a value type (meaning that it is not a reference), can be used as **rvalue** reference.

  This is rather subtle. We have already seen that values can be used as **const** reference. In that case, the value is stored in a temporary variable, after which a **const** reference to this variable is created. This happens for example when one writes

  ```
  std::ostream&
  operator << ( std::ostream& out, const string& s )
  { ... }

  std::cout << string( "hello, " ) + std::string( "world" ) << "\n";
  ```

In addition to being used a **const** reference, a temporary variable can also be used as **rvalue** reference. This makes sense, because the receiver is guaranteed to be the only user.

The rules for automatic creation of **rvalue** references automatically solve case 1 above. Case 3 can be solved by writing

```
template< typename T >
T exchange( T& t1, T&& t2 )
{
   auto old = std::move(t1);
   t1 = std::move(t2);
   return old;
}
```

Case 2 was solved internally by the implementation of `std::vector`. If the type of the vector has a copy-constructor that cannot fail, it will be used when resizing. This is the reason why one should write `noexcept` at the **rvalue** reference copy-constructor and assignment.

16. Implement `deque( deque&& q ) noexcept;` Actually, the implementation is already shown above. Make sure to put it in **deque.cpp**. It is possible to use only member initialization and `std::exchange`.

17. Implement `const deque& operator = ( deque&& q ) noexcept;` Clean up the heap data, copy the local data, and leave `q` in empty state. Here you can also use `std::exchange`. There is no need to check for self assignment, because the standard forbids it. It would make no sense for **rvalue** reference.

18. Put print statements in your **rvalue** CC and assignment, add this code:

    ```
    // Exchanges q and q2:

    q2 = std::exchange( q, std::move(q2) );
    ```

    Verify that **rvalue** reference CC and assignment are used.

5 Submission

Submit files **deque.cpp** and **main.cpp**. Make sure that **deque.cpp** contains no print statements, except in function `print` itself. Make sure that file **deque.cpp** contains no calls of `check_invariant( )` in the submitted version. Make sure that your **main.cpp** compiles with `valtype = double;` when submitted.