Design Patterns
Project Report
STL Containers in C

Sai Eashwar K S	Sreyans Bothra	Monisha Chandra
PES1201801910	PES120182012	PES1201802102

DEQUE

IMPLEMENTATION OVERVIEW:

Deque stands for doubly ended queue. The deque (of a particular type) we have implemented is defined as a simple structure with 5 fields.

Field_Name	Data Type	Usage
back	deque_node_##type*	Points to the last filled location
front	deque_node_##type*	Points to the first filled location
size	integer	Holds the number of elements in the deque.
maxie	integer	Holds the maximum number of elements that can be stored in the deque.
functions	Struct functions_deque_##type	Stores pointers to the allowable deque functions.

deque_node_##type -> a linked list like data structure to store the data.

Field_Name	Data Type	Usage
next	deque_node_##type*	Points to the next element in the deque
prev	deque_node_##type*	Points to the previous element in the deque
data	type	Holds the data part of the element.

CONTAINER PROPERTIES:

Sequence
Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.

Dynamic array
Generally implemented as a dynamic array, it allows direct access to any element in the sequence and provides relatively fast addition/removal of elements at the beginning or the end of the sequence.

MEMBER FUNCTIONS:

- Constructor: Can be constructed via construct deque container or copy constructor.
- Iterators:
    - begin -> points to the first element of the deque
    - end -> points to a location past the last element
    - rbegin -> points to the last element of the deque
    - rend -> points to a location previous to the first element
- Capacity:
    - size -> shows the current size of the array `queue`.
    - max_size -> shows the max_size of the array `queue`. We have given the value of 100000.
    - empty -> returns if the array `queue` is empty.
- Element Access:
    - at -> returns the element at the given valid location.
    - front -> returns the first element in the array `queue`
    - back -> returns the last element in the array `queue`
- Modifiers:
    - insert -> to insert at the end of the array `queue`
    - erase: Just clears the deque by making the size=0.
    - pop/remove -> removes the first element in the array `queue`.

USAGE

We have 2 header files with similar function definitions and declarations.

- `deque.h` -> is a linked list type implementation of a deque.
- `queue.h` -> stores the elements in an array storing elements of the given type.

Both these header files have `stdlib` and `stdarg` header files already included.

To use in a C file:

1) Just include the header file of your choice (either `queue.h` or `deque.h`) in your code.
2) In the global scope define the type of deque you need.
Example: define_queue(int): Where we are intending to use a deque storing elements of type int.
3) To use a deque:
- Declare a pointer as deque(type)* var_name. Example: deque(int)* a.
- Initialize it by calling ```new_deque(type,<argi>,<optional argument>)```.
    - Example :a=new_deque(int,5,6)-> initializes a deque with first 5 elements as 6
    - 'argi'
        - can be an integer to denote initial number of elements followed by the optional argument to represent which value needs to be stored.
        - can be a pointer to some other deque.
- Then other functions can be called as fun_name(pointer). Example: size(a)

LIST

IMPLEMENTATION OVERVIEW:

List is doubly linked list implementation allowing users to create doubly linked lists to store data of any simple type.

Implementation fields of the list :
(Note : “type” refers to the type of data the client wants to store)

Field_Name	Data Type	Usage
head	list_node_##type*	Points to the first element
tail	list_node_##type*	Points to the last element
size	Integer	Holds the number of elements in the list
functions	list_function_pointers_##type	Stores pointers to the allowable list functions

Implementation fields of node :

Field_Name	Data Type	Usage
data	type	Contains the data the user wants to store
next	list_node_##type*	Points to the next element
prev	list_node_##type*	Points to the previous element

Implementation fields of Iterator_list :

Field_Name	Data Type	Usage
iter	list_node_##type*	Points to an element in the list
is_reverse	Integer	Specifies if the iterator is forward or reverse

CONTAINER PROPERTIES:

Sequence
Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.

Doubly-linked list
Each element keeps information on how to locate the next and the previous elements, allowing constant time insert and erase operations before or after a specific element (even of entire ranges), but no direct random access.

MEMBER FUNCTIONS:

Constructor:

list(type)* new_list (type, 0)	Default constructor
list(type)* new_list (type, int n, type data)	Fill constructor Creates a new list with n elements having data
list(type)* new_list (type, list(type)*)	Copy constructor Creates a new list containing the elements of the passed list

Destructor:

void delete (list(type*))	Destructor

Iterators:

iterator_list(type)* begin (list(type)*)	Return iterator to the first element
iterator_list(type)* end (list(type)*)	Return iterator to NULL (conceptually beyond the last element)
iterator_list(type)* rbegin (list(type)*)	Return reverse iterator to the last element
iterator_list(type)* rend (list(type)*)	Return reverse iterator to NULL (conceptually beyond the first element)

Capacity:

int empty (const list(type)*)	Return true if list is empty, else returns false
int size (const list(type)*)	Return the number of elements in the list

Element access:

const type front (list(type)*)	Access first element
const type back (list(type)*)	Access last element
iterator_list(type)* find (list(type)*, type)	Returns iterator to the first element containing data, returns NULL if element not present
iterator_list(type)* find_if (list(type)*, int (*pred)(type))	Returns iterator to the first element satisfying the unary predicate, returns NULL if element not present

Modifiers:

void push_front (list(type)*, type)	Insert element at beginning
void push_back (list(type)*, type)	Insert element at end
void pop_front (list(type)*)	Delete first element
void pop_back (list(type)*)	Delete last element
void insert (list(type)*, iterator_list(type)* pos, int n, type data)	Insert n elements having data before pos
void clear (list(type*))	Clears all the elements and size becomes 0

Operations:

void remove_list (list(type)*, type)	Remove all elements with specific value
void remove_if_list (list(type)*, int (*pred)(type))	Remove elements satisfying condition
void reverse (list(type)*)	Reverse the list
void sort (list(type)*)	Sort the list
void sort_by (list(type)*, int(*pred)(type, type)	Sort the list based on the binary predicate

Methods on Iterator:

type* iter_list_deref (iterator(list)*)	Returns data of the element pointed to by iterator
int iter_list_equal (iterator(list)*, iterator(list)*)	Returns true if both iterators point to the same element, else false
int iter_list_notequal (iterator(list)*, iterator(list)*)	Return true if both iterators do not point to the same element, else false
iterator(list)* iter_list_forward (iterator(list)*)	Return the iterator to next element
iterator(list)* iter_list_backward (iterator(list)*)	Return the iterator to previous element

EXECUTION

Instructions for client file :

- Include list.h
- To use a list for a particular type, the list has to be defined first before main
- format to define list : define_list(type)

    Eg: define_list(int); define_list(char);

Execution :

Compile and execute the client file

Ex :

```
$ gcc client.c
$ ./a.out
```

VECTOR

IMPLEMENTATION OVERVIEW:

Vectors are sequence containers representing dynamic arrays with the ability to resize itself automatically.

Implementation fields of the list :
(Note : “type” refers to the type of data the client wants to store)

Field_Name	Data Type	Usage
size	Integer	Number of elements in the vector
capacity	Integer	Size of the storage space currently allocated for the vector, expressed in terms of elements
data	type *	An array to store elements
functions	vector_function_pointers_##type	Stores pointers to the allowable vector functions

Implementation fields of Iterator_vector :

Field_Name	Data Type	Usage
iter	type*	Points to an element in the vector
is_reverse	Integer	Specifies if the iterator is forward or reverse

CONTAINER PROPERTIES:

Sequence
Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.

Dynamic array
Allows direct access to any element in the sequence, even through pointer arithmetics, and provides relatively fast addition/removal of elements at the end of the sequence.

MEMBER FUNCTIONS:

Constructor:

vector(type)* new_vector (type, 0)	Default constructor
vector(type)* new_vector (type, int n, type data)	Fill constructor Creates a new vector with n elements having data
vector(type)* new_vector (type, vector(type)*)	Copy constructor Creates a new vector containing the elements of the passed vector

Destructor:

void delete (vector(type*))	Destructor

Iterators:

iterator_vector(type)* begin (vector(type)*)	Return iterator to the first element
iterator_vector(type)* end (vector(type)*)	Returns an iterator referring to the past-the-end element in the vector.
iterator_vector(type)* rbegin (vector(type)*)	Return reverse iterator to the last element
iterator_vector(type)* rend (vector(type)*)	Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector

Capacity:

int empty (const vector(type)*)	Return true if vector is empty, else returns false
int size (const vector(type)*)	Return the number of elements in the vector
int capacity (const vector(type)*)	Returns the size of the storage space currently allocated for the vector, expressed in terms of elements.

Element access:

const type front (vector(type)*)	Access first element
const type back (vector(type)*)	Access last element
iterator_vector(type)* find (vector(type)*, type)	Returns iterator to the first element containing data, returns end of vector if not found

Modifiers:

void push_back (vector(type)*, type)	Insert element at end
void pop_back (vector(type)*)	Delete last element
void insert (vector(type)*, iterator_vector(type)* it, type data)	Insert element having data at the iterator
iterator_vector(type)* erase (vector(type)*, type, iterator_vector(type)* it1, iterator_vector(type)* it2)	Removes from the vector a range of elements [it1,it2). Returns an iterator pointing to the new location of the element that followed the last element erased
void clear (vector(type*))	Clears all the elements and size becomes 0

Operations:

void at (vector(type)*, int)	Returns value of element at that position in the vector.
void reserve (vector(type)*, int)	Requests that vector capacity be at least enough to contain given no. of elements

Methods on Iterator:

type* iter_vector_deref (iterator(vector)*)	Returns element data at the iterator
int iter_vector_equal (iterator(vector)*, iterator(vector)*)	Returns true if both iterators point to the same element, else false
int iter_vector_notequal (iterator(vector)*, iterator(vector)*)	Return true if both iterators do not point to the same element, else false
iterator(vector)* iter_vector_forward (iterator(vector)*)	Return the iterator to next element
iterator(vector)* iter_vector_backward (iterator(vector)*)	Return the iterator to previous element
iterator(vector)* iter_vector_add (iterator(vector)*, int)	Return resulting iterator after adding an integer number to a given iterator
iterator(vector)* iter_vector_subtract (iterator(vector)*, int)	Return resulting iterator after adding an integer number to a given iterator
int iter_vector_lesser (iterator(vector)*, iterator(vector)*)	Returns true if first iterator is lesser than the second one, else false
int iter_vector_greater (iterator(vector)*, iterator(vector)*)	Returns true if first iterator is greater than the second one, else false
int iter_vector_lesser_equal (iterator(vector)*, iterator(vector)*)	Returns true if first iterator is lesser than or equal to the second one, else false
int iter_vector_greater_equal (iterator(vector)*, iterator(vector)*)	Returns true if first iterator is greater than or equal to the second one, else false

EXECUTION

Instructions for client file :

- Include vector.h
- To use a vector for a particular type, the vector has to be defined first before main
- format to define vector : define_vector(type)

    Eg: define_vector(int); define_vector(char);

Execution :

Compile and execute the client file

Ex :

```
$ gcc client.c
$ ./a.out
```