Design Patterns Project Report

Project Title: Implementation of Generic Programming in C
Project ID: 4

Team Details:

Darshan D - PES1201801456
Karan Kumar G - PES1201801883
Mayur P L - PES1201801439

Abstract:

Implementation of a few generic programming features in C.
We have implemented a few generic containers - list, stack, queue, vector and hashmap.
We have implemented iterators for each of these containers.
We have also implemented a few generic algorithms that make use of a pair of iterators.

Implementation Details:

The generic programming features have been implemented as a header file that the client can include in their program. The implementation of the generic programming features has been done majorly through the use of preprocessor directives.

GENERIC CONTAINERS:

The generic containers that have been implemented are as follows:

- ## LIST

  List has been implemented as a doubly linked list.
  The structure representing a node of the list includes the following attributes:
  - Key - The type of this key is generic and is decided by the user at compile time
  - Pointer to the next node in the list
  - Pointer to the previous node in the list

  The structure representing the list includes the following attributes:
  - Pointer to the head of the list
  - Pointer to the tail of the list

- An implementation field signifying the type of the key
- Function pointers to all the functions that are supported by list

The operations that can be performed on list are as follows:

- insert_at_end: This allows the client to insert an element at the end of the list
- insert_at_beg: This allows the client to insert an element at the beginning of the list
- delete_from_beg: This allows the client to delete an element from the beginning of the list
- delete_from_end: This allows the client to delete an element from the end of the list
- delete_key: This allows the client to delete a particular key from the list
- disp_list: This allows the client to display the contents of the list

Usage Details:

- The client has to first create a type for the particular type of list.
  Ex: To create the list of double type, the client will have to use LIST(double_l, double).
  The client can then create a variable of this type as follows:
  - double_l l1

  The client will have to initialize the list as follows:
  - init_list(double, l1)

  Once this is done, the client can perform different operations on the list by calling the appropriate functions and by passing the required arguments.

List Iterator:

The structure representing the list iterator includes the following attributes:

- Pointer to the node of the list
- An implementation field signifying the type of the key
- Function pointers to the functions that are supported by the iterator

The list iterator has been implemented as a **forward iterator** as it supports the following operations:

- equality: This is used to compare two iterators for equality
- inequality: This is used to compare two iterators for inequality
- get_value: This is used to get the key of the node pointed to by the iterator
- set_current: This is used to set the iterator to point to the node as pointed to by another iterator
- next: This is used to increment the iterator to point to the next node
- has_next: This is used to check whether the iterator can be incremented

- advance: This is used to advance the iterator by said distance

Apart from this, the following two functions have also been defined:
- begin: This creates an iterator pointing to the first node in the list
- end: This creates an iterator pointing to NULL

Usage Details:
Ex: Assuming double_l is the list of double type created, the corresponding iterator can be created as follows:
- double_l_iterator it_l_d

The iterator can be initialized as follows:
- init_list_iterator(double, l1, it_l_d)

Once the iterator is initialized, the client can perform different operations on the iterator by calling the appropriate functions and by passing the required arguments.

● STACK

Stack has been implemented as a doubly linked list.
The structure representing a node of the stack includes the following attributes:
- Key - The type of this key is generic and is decided by the user at compile time
- Pointer to the next node in the stack
- Pointer to the previous node in the stack

The structure representing the stack includes the following attributes:
- Pointer to the head of the stack
- Pointer to the top of the stack
- An implementation field signifying the type of the key
- Function pointers to all the functions that are supported by stack

The operations that can be performed on stack are as follows:
- push: This allows the client to insert an element at the end of the stack
- pop: This allows the client to delete an element from the end of the stack
- disp_stack: This allows the client to display the contents of the stack
- peek_stack: This allows the client to retrieve the key at the top of the stack

Usage Details:
- The client has to first create a type for the particular type of stack.

Ex: To create the stack of int type, the client will have to use STACK(int_s, int).
The client can then create a variable of this type as follows:
- int_s s1

The client will have to initialize the stack as follows:
- init_stack(int, s1)

Once this is done, the client can perform different operations on the stack by calling the appropriate functions and by passing the required arguments.

Stack Iterator:

The structure representing the stack iterator includes the following attributes:
- Pointer to the node of the stack
- An implementation field signifying the type of the key
- Function pointers to the functions that are supported by the iterator

The stack iterator has been implemented as a **forward iterator** as it supports the following operations:
- equality: This is used to compare two iterators for equality
- inequality: This is used to compare two iterators for inequality
- get_value: This is used to get the key of the node pointed to by the iterator
- set_current: This is used to set the iterator to point to the node as pointed to by another iterator
- next: This is used to increment the iterator to point to the next node
- has_next: This is used to check whether the iterator can be incremented
- advance: This is used to advance the iterator by said distance

Apart from this, the following two functions have also been defined:
- begin: This creates an iterator pointing to the first node in the stack
- end: This creates an iterator pointing to NULL

Usage Details:

Ex: Assuming int_s is the stack of int type created, the corresponding iterator can be created as follows:
- int_s_iterator it_s_d

The iterator can be initialized as follows:
- init_stack_iterator(int, s1, it_s_d)

Once the iterator is initialized, the client can perform different operations on the iterator by calling the appropriate functions and by passing the required arguments.

- # QUEUE

Queue has been implemented as a doubly linked list.
The structure representing a node of the queue includes the following attributes:

- Key - The type of this key is generic and is decided by the user at compile time
- Pointer to the next node in the queue
- Pointer to the previous node in the queue

The structure representing the queue includes the following attributes:

- Pointer to the front of the queue
- Pointer to the tail of the queue
- An implementation field signifying the type of the key
- Function pointers to all the functions that are supported by queue

The operations that can be performed on queue are as follows:

- enqueue: This allows the client to insert an element at the end of the queue
- dequeue: This allows the client to delete an element from the front of the queue
- disp_queue: This allows the client to display the contents of the queue
- peek_queue: This allows the client to retrieve the key at the front of the queue

Usage Details:

- The client has to first create a type for the particular type of queue.
  Ex: To create the queue of int type, the client will have to use QUEUE(int_q, int).
  The client can then create a variable of this type as follows:
  - int_q q1

  The client will have to initialize the queue as follows:
  - init_queue(int, q1)

  Once this is done, the client can perform different operations on the queue by calling the appropriate functions and by passing the required arguments.

Queue Iterator:

The structure representing the queue iterator includes the following attributes:

- Pointer to the node of the queue
- An implementation field signifying the type of the key
- Function pointers to the functions that are supported by the iterator

The queue iterator has been implemented as a **forward iterator** as it supports the following operations:

- equality: This is used to compare two iterators for equality
- inequality: This is used to compare two iterators for inequality
- get_value: This is used to get the key of the node pointed to by the iterator
- set_current: This is used to set the iterator to point to the node as pointed to by another iterator
- next: This is used to increment the iterator to point to the next node
- has_next: This is used to check whether the iterator can be incremented
- advance: This is used to advance the iterator by said distance

Apart from this, the following two functions have also been defined:

- begin: This creates an iterator pointing to the first node in the queue
- end: This creates an iterator pointing to NULL

Usage Details:
Ex: Assuming int_q is the queue of int type created, the corresponding iterator can be created as follows:

- int_q_iterator it_q_d

The iterator can be initialized as follows:

- init_queue_iterator(int, q1, it_q_d)

Once the iterator is initialized, the client can perform different operations on the iterator by calling the appropriate functions and by passing the required arguments.

- ## VECTOR

Vector has been implemented using a dynamic table.
Vector allows for random access using an index.
The structure representing a node of the vector includes the following attributes:

- An attribute signifying the size of the dynamic table
- An attribute signifying the number of filled records in the dynamic table
- An attribute having the dynamic table created

The structure representing the vector includes the following attributes:

- Pointer to the node of the vector
- An implementation field signifying the type of the key
- Function pointers to all the functions that are supported by vector

The operations that can be performed on vector are as follows:

- push_back: This allows the client to insert an element into the vector. It inserts a record at the end of the dynamic table associated with the vector
- pop_back: This allows the client to delete the last record from the dynamic table associated with the vector
- access: This allows the client to access a particular element of the vector based on the index. It allows for random access.
- my_free: This allows the client to free the dynamic table associated with the vector
- disp_vector: This allows the client to display the contents of the vector.

Usage Details:

- The client has to first create a type for the particular type of vector.
  Ex: To create the vector of int type, the client will have to use VECTOR(int_v, int).
  The client can then create a variable of this type as follows:
  - int_v v1

  The client will have to initialize the vector as follows:
  - init_vector(int, 100, v1)

  Once this is done, the client can perform different operations on the vector by calling the appropriate functions and by passing the required arguments.

Vector Iterator:

The structure representing the vector iterator includes the following attributes:

- Pointer to the node of the vector
- An implementation field signifying the current index
- An implementation field signifying the type of the key
- Function pointers to the functions that are supported by the iterator

The vector iterator has been implemented as a **forward iterator** as it supports the following operations:

- equality: This is used to compare two iterators for equality
- inequality: This is used to compare two iterators for inequality
- get_value: This is used to get the key of the record pointed to by the iterator
- set_current: This is used to set the iterator to point to the record as pointed to by another iterator
- next: This is used to increment the iterator to point to the next record in the vector

- has_next: This is used to check whether the iterator can be incremented
- advance: This is used to advance the iterator by said distance

Apart from this, the following two functions have also been defined:

- begin: This creates an iterator pointing to the first record in the dynamic table associated with the vector
- end: This creates an iterator pointing to NULL

Usage Details:
Ex: Assuming int_v is the vector of int type created, the corresponding iterator can be created as follows:

- int_v_iterator it_v_d

The iterator can be initialized as follows:

- init_vector_iterator(int, v1, it_v_d)

Once the iterator is initialized, the client can perform different operations on the iterator by calling the appropriate functions and by passing the required arguments.

● HASHMAP

Hashmap has been implemented using an array of structures, hashing and separate chaining to handle collisions during hashing.

Hashmap allows the client to map a key to a value.

The structure representing a node of the hashmap includes the following attributes:

- An attribute signifying the type of key
- An attribute signifying the type of value
- Pointer to the next node in the hashmap

The structure representing the hashmap includes the following attributes:

- An attribute signifying the size of the hashmap
- An implementation field signifying the type of the key
- An implementation field signifying the type of the value
- Pointer to the array associated with the hashmap
- Function pointers to all the functions that are supported by hashmap

The operations that can be performed on hashmap are as follows:

- insert_map: This allows the client to insert a key-value pair into the hashmap
- delete_map: This allows the client to delete a particular key and the value associated with that key from the hashmap
- retrieve_map: This allows the client to retrieve the value associated with a particular key, if it exists
- disp_map: This allows the client to display the contents of the hashmap

Usage Details:

- The client has to first create a type for the particular type of hashmap.
  Ex: To create the hashmap that takes the key of double type and value of int type , the client will have to use MAP(double_int_m, double, int).
  The client can then create a variable of this type as follows:
  - double_int_m m1

  The client will have to initialize the hashmap of size 100 as follows:
  - init_map(double, int, 100, m1)

  Once this is done, the client can perform different operations on the hashmap by calling the appropriate functions and by passing the required arguments.

Hashmap Iterator:

The structure representing the hashmap iterator includes the following attributes:

- Pointer to the node of the hashmap
- An implementation field signifying the current index
- An implementation field signifying the size of the array associated with the hashmap
- An implementation field signifying the type of the key
- An implementation field signifying the type of the value
- Function pointers to the functions that are supported by the iterator

The hashmap iterator has been implemented as a **forward iterator** as it supports the following operations:

- equality: This is used to compare two iterators for equality
- inequality: This is used to compare two iterators for inequality
- get_value: This is used to get the key of the node pointed to by the iterator
- set_current: This is used to set the iterator to point to the node as pointed to by another iterator
- next: This is used to increment the iterator to point to the next valid node in the array associated with the hashmap

- has_next: This is used to check whether the iterator can be incremented
- advance: This is used to advance the iterator by said distance

Apart from this, the following two functions have also been defined:

- begin: This creates an iterator pointing to the first valid node in the array associated with the hashmap
- end: This creates an iterator pointing to NULL

Usage Details:
Ex: Assuming double_int_m is the hashmap created, having keys of double type and values of int type, the corresponding iterator can be created as follows:

- double_int_m_iterator it_m_d

The iterator can be initialized as follows:

- init_map_iterator(double, int, m1, it_m_d)

Once the iterator is initialized, the client can perform different operations on the iterator by calling the appropriate functions and by passing the required arguments.

GENERIC ALGORITHMS:

The Generic algorithms that have been implemented are as follows:

- **Find:** Find takes in a pair of iterators(first and last) and the key to be found. It moves the first iterator until it points to the required key. If the required key is not found in the range mentioned, the first iterator will be pointing to the last iterator indicating that the key is not found.

- **Find_if:** Find_if takes in a pair of iterators(first and last) and the predicate (a function pointer). It moves the first iterator until it points to the key that satisfies the predicate. If a key satisfying the predicate is not found in the range mentioned, the first iterator will be pointing to the last iterator indicating that a key satisfying the predicate has not been found.

- **Count:** Count takes in a pair of iterators(first and last), key whose count is to be found and the variable that will store the count. It traverses the container until the first iterator points to the last iterator and counts the number of times the given key occurs in this range.

- **Count_if:** Count takes in a pair of iterators(first and last), predicate(function pointer) and the variable that will store the count. It traverses the container

until the first iterator points to the last iterator and counts the number of times a key satisfying the predicate occurs in this range.

- **Min:** Min takes in a pair of iterators(first and last), a variable(min) that will hold the minimum value and a flag variable denoting if the min variable has been set to a valid value. It traverses the container until the first iterator points to the last iterator and finds the minimum key that occurs in this range. If the first and last iterator point to the same node, it means that the range of the container to be traversed is empty, then the flag variable is set to 0 indicating that the value of the min variable is garbage and does not hold the actual min value. Otherwise the min variable is set to the appropriate value and the flag is set to 1.

- **Max:** Max takes in a pair of iterators(first and last), a variable(max) that will hold the maximum value and a flag variable denoting if the max variable has been set to a valid value. It traverses the container until the first iterator points to the last iterator and finds the maximum key that occurs in this range. If the first and last iterator point to the same node, it means that the range of the container to be traversed is empty, then the flag variable is set to 0 indicating that the value of the max variable is garbage and does not hold the actual max value. Otherwise the max variable is set to the appropriate value and the flag is set to 1.

- **Accumulate:** Accumulate takes in a pair of iterators(first and last) and a variable(acc) that holds the accumulated value. It traverses the container until the first iterator points to the last iterator, accumulating the keys that are found in this range. The accumulated value is stored in the acc variable.