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

Team Details:

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

Interfaces Provided:

LIST

- **List:**
  - Function Signature: LIST(alias, TYPE)
  - Usage:
    - LIST(int_l, int) // In the global namespace
    - int_l l1;
  - Creates a list with keys of type 'TYPE'. The alias is then used to declare and initialise variables of this list type. Also initialises iterator type for the same type as alias_iterator. alias_iterator can be then used to create iterators for this type.

- **Initialise List:**
  - Function Signature: init_list(TYPE, alias list)
  - Usage: init_list(int, l1)
  - Initialises the list. Here TYPE is one of the primitive types. Passes the objects by reference.

- **Insert at beginning:**
  - Function Signature: insert_at_beg(int_l *l1, TYPE key)
  - Return type: void
  - Usage: l1.insert_at_beg(&l1, 5)

- **Insert at end:**
  - Function Signature: insert_at_end(int_l *l1, TYPE key)
  - Return type: void
  - Usage: l1.insert_at_end(&l1, 5)

- **Delete from beginning:**
  - Function Signature: delete_from_beg(int_l *l1)
  - Return type: void
  - Usage: l1.delete_from_beg(&l1)

- **Delete from end:**
  - Function Signature: delete_from_end(int_l *l1)
  - Return type: void
  - Usage: l1.delete_from_end(&l1)

- **Delete key:**
  - Function Signature: delete_key(int_l *l1, TYPE key)
  - Return type: void
  - Usage: l1.delete_key(&l1, 5)

- **Display list:**
  - Function Signature: disp_list(const int_l *l1)
  - Return type: void
  - Usage: l1.disp_list(&l1)

- **Init_list_iterator:**
  - Function Signature: init_list_iterator(TYPE, list, iterator)
  - Usage:
    - int_l_iterator it_l_i
    - init_list_iterator(int, l1, it_l_i)

STACK

- **Stack:**
  - Function Signature: STACK(alias, type)
  - Usage:
    - STACK(double_s, double) // In the global namespace
    - double_s s1;
  - Creates a stack with keys of type ‘type’. The alias is then used to declare and initialise variables of this stack type. Also initialises iterator type for the same type as alias_iterator. alias_iterator can be then used to create iterators for this type.

- **Initialise Stack:**
  - Function Signature: init_stack(type, alias stack)
  - Usage: init_stack(double, s1)
  - Initialises the stack. Here type is one of the primitive types. Passes the objects by reference.

- **Push**
  - Function Signature: push(double_s* s1, TYPE key)
  - Return type: void

    - Usage: s1.push(&s1, 5.0)

- **Pop**
    - Function Signature: pop(double_s* s1)
    - Return type: void
    - Usage: s1.pop(&s1)

- **Peek**
    - Function Signature: peek_stack(const double_s* s1)
    - Return type: TYPE
    - Usage: s1.peek_stack(&s1)

- **Display stack**
    - Function Signature: disp_stack(const double_s* s1)
    - Return type: void
    - Usage: s1.disp_stack(&s1)

- **Init_stack_iterator**
    - Function Signature: init_stack_iterator(TYPE, stack, iterator)
    - Usage:
        - double_s_iterator it_s_i
        - init_stack_iterator(double, s1, it_s_i)

QUEUE

- **Queue:**
    - Function Signature: QUEUE(alias, type)
    - Usage:
        - QUEUE(double_q, double) // In the global namespace
        - double_q q1;
    - Creates a queue with keys of type ‘type’. The alias is then used to declare and initialise variables of this queue type. Also initialises iterator type for the same type as alias_iterator. alias_iterator can be then used to create iterators for this type.

- **Initialise queue**:
    - Function Signature: init_queue(type, alias stack)
    - Usage: init_queue(double, q1)
    - Initialises the queue. Here type is one of the primitive types. Passes the objects by reference.

- **enqueue**
    - Function Signature: enqueue(double_q* ptr_q, TYPE key)
    - Return type: void
    - Usage: q1.enqueue(&q1, 5.0)

- **dequeue**
    - Function Signature: dequeue(double_q* ptr_q)
    - Return type: void
    - Usage: q1.dequeue(&q1)

- **Display queue**
    - Function Signature: disp_queue(const double_q* ptr_q)
    - Return type: void
    - Usage: q1.disp_queue(&q1)

- **Peek**
    - Function Signature: peek_queue(const double_q* q1)
    - Return type: TYPE
    - Usage: q1.peek_queue(&q1)

- **Init_queue_iterator**
    - Function Signature: init_queue_iterator(TYPE, queue, iterator)
    - Usage:
        - double_q_iterator it_q_d
        - init_queue_iterator(double, q1, it_q_d)

VECTOR

- **Vector:**
    - Function Signature: VECTOR(alias, type)
    - Usage:
        - VECTOR(char_v, char) // In the global namespace
        - char_v v1;
    - Creates a vector with keys of type ‘type’. The alias is then used to declare and initialise variables of this vector type. Also initialises iterator type for the same type as alias_iterator. alias_iterator can be then used to create iterators for this type.

- **Initialise Vector:**
    - Function Signature: init_vector(type, alias vector)
    - Usage: init_vector(char, 100 ,v1)
    - Initialises the vector. Here type is one of the primitive types. Passes the objects by reference.

- **Pushback**
    - Function Signature: push_back(char_v* v1, TYPE key)
    - Return type: void
    - Usage: v1.push_back(&v1, ‘F’)

- **Popback**
    - Function Signature: pop_back(char_v* v1)
    - Return type: void
    - Usage: v1.pop(&v1)

- **Access**
    - Function Signature: access(const char_v* v1, int index)
    - Return type: TYPE
    - Usage: v1.access(&v1, 0)

- **Display vector**
    - Function Signature: disp_vector(const char_v* v1)
    - Return type: void
    - Usage: v1.disp_vector(&v1)

- **Init_vector_iterator**
    - Function Signature: init_vector_iterator(TYPE, vector, iterator)
    - Usage:
        - char_v_iterator it_v_i
        - init_vector_iterator(char, v1, it_v_i)

HASHMAP

- **Hashmap**:
    - Function Signature: MAP(alias, type_key, type_val)
    - Usage:
        - MAP(double_int_m, double, int) // In the global namespace
        - double_int_m v1;
    - Creates a map with keys of type “type_key” and values of type “type_val”. The alias is then used to declare and initialise variables of this map type. Also initialises iterator type for the same type as alias_iterator. alias_iterator can be then used to create iterators for this type.

- **Initialise Hashmap:**
    - Function Signature: init_map(type_key, type_value, size, alias map)

    - Usage: init_map(int, int, 100, l1)
    - Initialises the map. Passes the objects by reference.

- **Insert_map**
    - Function Signature: insert_map(char_m* m1, TYPE key, TYPE value)
    - Return type: void
    - Usage: m1.insert_map(&m1, ‘F’, 20.9)

- **Delete_map**
    - Function Signature: delete_map(char_m* m1, TYPE key)
    - Return type: void
    - Usage: m1.delete_map(&m1, 1)

- **Retrieve_map**
    - Function Signature: retrieve_map(char_m* m1, TYPE key, int* exists)
    - Return type: TYPE_val
    - Usage: m1.retrieve_map(&v1, “key”, ptr)

- **Display map**
    - Function Signature: disp_map(const char_m* m1)
    - Usage: v1.disp_map(&m1)

- **Init_map_iterator**
    - Function Signature: init_map_iterator(TYPE_key, TYPE_value, map, iterator)
    - Usage:
        - char_m_iterator it_m_i
        - init_map_iterator(char, int, m1, it_m_i)

Generalised Iterator Interfaces provided:

- **Has Next**
    - Function Signature: int has_next(iterator*)

- **Next**
    - Function Signature: TYPE next(iterator*)

- **Get Value**
    - Function Signature: TYPE get_value(iterator*)

- **Equality**
    - Function Signature: int equality(iterator*)

- **Inequality**
    - Function Signature: int inequality(iterator*)

- **Advance**
    - Function Signature: void advance(iterator*)

Generic Algorithms:

- **Find**:
    - Function Signature: find(first, last, key)

- **Find_if**
    - Function Signature: find_if(first, last, pred)

- **Count**
    - Function Signature: count(first, last, key, count)

- **Count_if**
    - Function Signature: count_if(first, last, pred, count)

- **Min**
    - Function Signature: min(first, last, min, flag)

- **Max**
    - Function Signature: max(first, last, max, flag)

- **Accumulate**
    - Function Signature: accumulate(first, last, acc)

Details about running the software:

The above generic programming features have been implemented as a header file that the client can include in their programs:
#include “generics.h”

Multiple client files have been provided each checking the working of a particular container:

- **test_list.c** : This test file checks the working and correctness of all functionalities provided for the “list” container and its iterator, along with the multiple algorithms implemented.

- **test_stack.c** : This test file checks the working and correctness of all functionalities provided for the “stack” container and its iterator, along with the multiple algorithms implemented.

- **test_queue.c** : This test file checks the working and correctness of all functionalities provided for the “queue” container and its iterator, along with the multiple algorithms implemented.

- **test_vector.c** : This test file checks the working and correctness of all functionalities provided for the “vector” container and its iterator, along with the multiple algorithms implemented.

- **test_map.c** : This test file checks the working and correctness of all functionalities provided for the “map” container and its iterator, along with the multiple algorithms implemented.

The client can run the client file using the gcc utility as follows:
gcc client.c -lm -o exec

The client can then load and execute the executable as follows:
./exec