Object Oriented Programming (CT-260)
Lab 11

Introduction to Exception Handling, C++ Standard Template Library (STL) and Vectors.

Objectives

The objective of this lab is to familiarize students C++ Introduction to Exception Handling, [illegible] students will be able to understand the concepts of Exception Handling along with use of STL, vector, set, and map classes using examples.

Tools Required

DevC++ IDE / Visual Studio / Visual Code

Course Coordinator –
Course Instructor –
Lab Instructor –
Prepared By Department of Computer Science and Information Technology
NED University of Engineering and Technology

Exception Handling in C++

The error handling mechanism of C++ is generally referred to as exception handling. C++ provides a mechanism of handling runtime errors in a program. Generally, exceptions are classified into synchronous and asynchronous exceptions.

Synchronous Exceptions: The exceptions which occur during the program execution (runtime) due to some fault in the input data or technique that is not suitable to handle the current class of data, within the program are known as synchronous exceptions. For example: errors such as out of range, division by zero, overflow, underflow and so on belong to the class of synchronous exceptions.

Asynchronous Exceptions: The exceptions caused by events or faults unrelated (external) to the program and beyond the control of the program are called asynchronous exceptions. For example: errors such as keyboard interrupts, hardware malfunctions, disk failure and so on belong to the class of asynchronous exceptions.

The exception handling mechanism of C++ is designed to handle only synchronous exceptions within a program. This is done by throwing an exception. The exception handling mechanism uses three blocks: ***try, throw*** and ***catch***. The try-block must be followed immediately by a *handler*, which is a catch block. If an exception is thrown in the try block, the program control is transferred to the appropriate exception handler. The program should attempt to catch any exception that is thrown by any function. Failure to do so may result in abnormal program termination.

Why Exception Handling?

The following are the main advantages of exception handling over traditional error handling:

1) ***Separation of Error Handling code from Normal Code***: In traditional error handling codes, there are always if-else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try/catch blocks, the code for error handling becomes separate from the normal flow.

2) ***Functions/Methods can handle only the exceptions they choose***: A function can throw many exceptions but may choose to handle some of them. The other exceptions, which are thrown but not caught, can be handled by the caller. If the caller chooses not to catch them, then the exceptions are handled by the caller of the caller. In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).

3) ***Grouping of Error Types***: In C++, both basic types and objects can be thrown as exceptions. We can create a hierarchy of exception objects, group exceptions in namespaces or classes and categorize them according to their types.

Exception Handling Constructs:

1. **throw**: The keyword throw is used to raise an exception when an error is generated in the computation.
2. **catch**: The exception handler is indicated by the catch keyword. It must be used immediately after the statements marked by the try keyword.
3. **try**: The try keyword defines the boundary within which an exception can occur.

Thus, the error handling code must perform the following tasks:

1. Detect the problem causing exception. (Will hit the exception).
2. Inform that an error has occurred. (Throw the exception).
3. Receive the error information (Catch the exception).
4. Take corrective actions. (Handle the exception).

The basic syntax for exception handling in C++ is given below:

```
try {

    // code that may raise an exception
    throw argument;
}

catch (exception) {
    // code to handle exception
}
```

The following is a simple example to show exception handling in C++.

```
#include <iostream>
using namespace std;

int main() {

    double numerator, denominator, answer;

    cout << "Enter numerator: ";
    cin >> numerator;

    cout << "Enter denominator: ";
    cin >> denominator;

    try {

        // throw an exception if denominator is 0
        if (denominator == 0)
            throw denominator;

        // not executed if denominator is 0
        answer = numerator / denominator;
        cout<<numerator<<"/"<<denominator<<"="<<divide<<endl;
    }

    catch (int num_exception) {
        cout<<"Error: Cannot divide by"<<num_exception<<endl;
    }

    return 0;
}
```

The above program divides two numbers and displays the result. But an exception occurs if the denominator is 0. To handle the exception, we have put the code `answer = numerator / denominator;` inside the try block. Now, when an exception occurs, the rest of the code inside the try block is skipped.

Output 1:

```
Enter numerator: 72
Enter denominator: 0
Error: Cannot divide by 0
```

Output 2:

```
Enter numerator: 72
Enter denominator: 3
72 / 3 = 24
```

The catch block catches the thrown exception and executes the statements inside it. If none of the statements in the try block generates an exception, the catch block is skipped.

denominator == 0

```
try {
    if (denominator == 0)
        throw denominator;
    // code
}
catch (int num){
      //code
}
return 0;
```

denominator != 0

```
try {
    if (denominator == 0)
        throw denominator;
    // code
}
catch (int num){
      //code
}
return 0;
```

Catching All Types of Exceptions

In exception handling, it is important that we know the types of exceptions that can occur due to the code in our try statement so that we can use the appropriate catch parameters. Otherwise, the try...catch statements might not work properly.

If we do not know the types of exceptions that can occur in our try block, there is a special catch block called the 'catch all' block, written as catch(...), that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so the catch(...) block will be executed.

```
#include <iostream>
using namespace std;

int main()
{
      try {
      throw 10;
      }
      catch (char excp) {
            cout << "Caught " << excp;
      }
      catch (...) {
            cout << "Default Exception\n";
      }
```

```
    return 0;
}
```

Since there is no exception for an int in the code, the default catch will be executed.

Output

```
Default Exception
```

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

C++ Multiple catch Statements

In C++, we can use multiple catch statements for different kinds of exceptions that can result from a single block of code.

```
try {
    // code
}
catch (exception1) {
    // code
}
catch (exception2) {
    // code
}
catch (...) {
    // code
}
```

The following program divides two numbers and stores the result in an array element. There are two possible exceptions that can occur in this program:

- If the index is out of bounds (index is greater than the size of the array)
- If a number is divided by 0

These exceptions are caught in multiple catch statements.

```
#include <iostream>
using namespace std;

int main() {

     double numerator, denominator, arr[4] = {0.0, 0.0, 0.0,
0.0};
    int index;

    cout << "Enter array index: ";
    cin >> index;

    try {
        // throw exception if array out of bounds
        if (index >= 4)
            throw "Error: Array out of bounds!";

        // not executed if array is out of bounds
        cout << "Enter numerator: ";
        cin >> numerator;
        cout << "Enter denominator: ";
```

```
        cin >> denominator;

        // throw exception if denominator is 0
        if (denominator == 0)
            throw denominator;

        // not executed if denominator is 0
        arr[index] = numerator / denominator;
        cout << arr[index] << endl;
    }

    // catch "Array out of bounds" exception
    catch (const char* msg) {
        cout << msg << endl;
    }

    // catch "Divide by 0" exception
    catch (int num) {
        cout << "Error: Cannot divide by " << num << endl;
    }

    // catch any other exception
    catch (...) {
        cout << "Unexpected exception!" << endl;
    }

    return 0;
}
```

Considering the following two cases, the outputs are explained in the text below.

Output 1:

```
Enter numerator: 72
Enter denominator: 0
Error: Cannot divide by 0
```

Output 2:

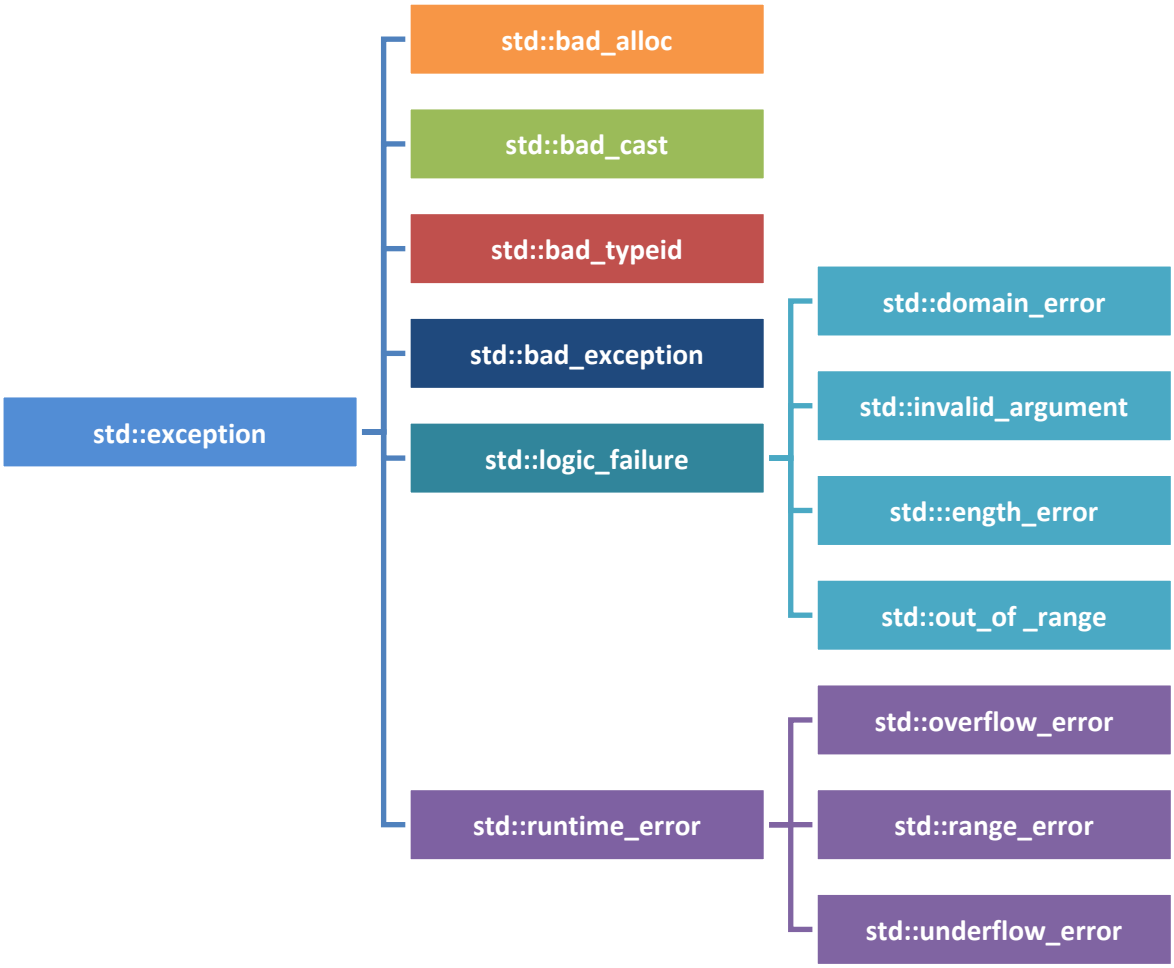
```
Enter numerator: 72
Enter denominator: 3
72 / 3 = 24
```

As the array arr only has 4 elements, the index cannot be greater than 3. In Output 1,index is 5. So we throw a string literal "`Error: Array out of bounds!`". This exception is caught by the first catch block. Notice the catch parameter const char* msg. This indicates that the catch statement takes a string literal as an argument.

In Output 2, the denominator is 0. So we throw the int denominator. This exception is caught by the second catch block. If any other exception occurs, it is caught by the default catch block.

C++ Standard Exception

In C++, the <stdexcept> header defines a set of standard exception types that are commonly used to handle errors and exceptional situations. Some of the standard exceptions in C++ are listed below:



These are some of the standard exceptions provided by C++. They help categorize and handle different types of errors and provide meaningful information about exceptional situations. They can be used by including the <stdexcept> header as shown in the example below.

```
#include <iostream>
#include <stdexcept>
using namespace std;

void divide(int numerator, int denominator) {
try{
    if (denominator == 0) {
        throw runtime_error("Division by zero error");
          }
    cout<<"The result is : "<<float(numerator)/denominator;
    }

catch (const exception& ex) {
        cout<<"Exception caught: "<<ex.what()<<endl;
    }
}

int main() {
    divide(10, 0);
    divide(10, 20);
    return 0;
```

Define New Exceptions

You can define your own exceptions standalone or by inheriting and overriding exception class functionality. Following is the example, which shows how you can define your own exception class.

```
#include <iostream>
using namespace std;

class MyException {
    public:
        const char * what()
        {
            return "Attempted to divide by zero!\n";
        }
};

int main()  {
    try{
        int x, y;
        cout << "Enter the two numbers : \n";
        cin >> x >> y;
        if (y == 0)  {
            MyException z;
            throw z;
        }
        else {
            cout << "x / y = " << x/y << endl;
        }
    }
    catch(MyException& e)  {
        cout << e.what();
    }
}
```

This code will produce the following output.

Output:
```
Attempted to divide by 0!
```

Here, what( ) is a public method provided by MyException class. This returns the cause of an exception.

The C++ Standard Template Library (STL)

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized. Working knowledge of template classes is a prerequisite for working with STL.

The C++ Standard Template Library (STL) is a collection of algorithms, data structures, and other components that can be used to simplify the development of C++ programs. The STL provides a range of containers, such as vectors, lists, and maps, as well as algorithms for searching, sorting and manipulating data. One of the key benefits of the STL is that it provides a way to write generic, reusable code that can be applied to different data types. This means that you can write an algorithm once, and then use it with different types of data without having to write separate code for each type. The STL also provides a way to

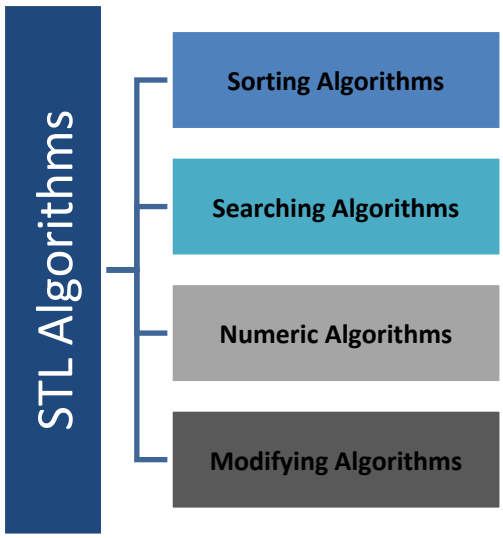
write efficient code. Many of the algorithms and data structures in the STL are implemented using optimized algorithms, which can result in faster execution times compared to custom code. C++ STL has 3 major components:

1. Algorithms
2. Containers
3. Iterators

In addition to these, STL also provides several other features, including function objects, smart pointers, and exception handling mechanisms.

C++ STL Algorithms

In C++, the Standard Template Library (STL) provides a rich set of algorithms that operate on containers or sequences of elements. These algorithms are generic and can be used with various container types. They offer a wide range of functionality, including sorting, searching, transforming, and modifying elements. Some of the most commonly used STL algorithms in C++ are shown in the following diagram.



These are just a few examples of the many algorithms available in the STL. These algorithms are designed to work efficiently with different container types, providing a powerful toolset for manipulating and processing data. To use these algorithms, include the <algorithm> header and make use of the appropriate algorithm by providing the necessary arguments and range iterators.

C++ STL Containers

The C++ Standard Library (STL) provides a rich set of container classes that offer different data structures for storing and manipulating collections of objects. Here are some commonly used STL containers in C++.

a. Sequence Containers:

In C++, the Standard Template Library (STL) provides several sequential containers that store and manage elements in a sequential manner. Some examples include vector (dynamic array class), linkedlist, stack, queues, etc. These sequential containers differ in their underlying data structures, performance characteristics, and supported operations. Depending on your requirements, you can choose the appropriate sequential container that suits your needs for element access, insertion, deletion, or traversal.

b. Associative containers:

In C++, the Standard Template Library (STL) provides several associative containers that store and manage elements in an associative manner, allowing efficient lookup and retrieval based on a key. The associative containers in C++ are set, multiset, map, and

multimap. These associative containers offer efficient lookup and retrieval operations based on keys. They provide different performance characteristics and trade-offs based on the underlying data structures used, such as binary search trees for ordered containers or hash tables for unordered containers. You can choose the appropriate associative container based on your requirements for fast access, unique or duplicate keys, or sorted order.

c. Unordered Associative Containers in C++

In C++, the Standard Template Library (STL) provides unordered associative containers that store and manage elements using hash-based data structures for efficient lookup and retrieval. These containers are implemented as hash tables and provide fast average-case performance for insertion, deletion, and search operations. The examples include unordered multiset, unordered multimap, etc. These unordered associative containers provide constant-time average-case complexity for insertion, deletion, and search operations, making them suitable for scenarios where fast lookup based on a key is required. The actual performance may vary depending on the quality of the hash function and the number of elements in the container.

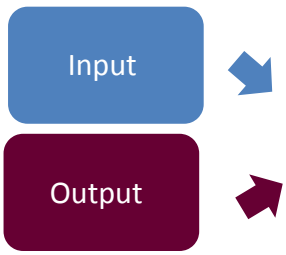
C++ STL ITERATOR

In C++, iterators are used to traverse and access elements in various data structures, including containers and sequences. They provide a uniform interface to work with different types of data structures, allowing you to iterate over elements, access values, and perform operations. Iterators are pointer-like entities used to access the individual elements in a container. Iterators are moved sequentially from one element to another element. This process is known as iterating through a container. Iterator contains mainly two functions:

begin( ): The member function begin( ) returns an iterator to the first element of the vector.
end( ): The member function end( ) returns an iterator to the past-the-last element of a container.

Iterator Categories

Iterators are mainly divided into five categories:



The C++ Standard Template Library (STL) provides different types of iterators with varying levels of functionality and capabilities. STL algorithms and containers often use iterators as arguments to perform operations on the underlying elements. For example, algorithms like std::sort and std::find accept iterators to specify the range of elements to operate on. Containers like std::vector and std::list provide member functions to obtain iterators for iterating over their elements.

Vectors

In C++, vectors are like dynamic arrays which are used to store elements of similar data types. However, unlike arrays, the size of a vector can grow dynamically. That is, we can change the size of the vector during the execution of a program as per our requirements.

The Standard Template Library (STL) provides a container called std::vector that represents a dynamic array. It is one of the most commonly used containers in C++ due to its flexibility, efficiency, and ease of use. To use vectors, we need to include the <vector> header file in our program.

C++ Vector Declaration

Once we include the header file, here's how we can declare a vector in C++:

```
vector<T> vector name;
```

The type parameter <T> specifies the type of the vector. It can be any primitive data type such as int, char, float, etc.

```
vector<int> num;
```

Notice that we have not specified the size of the vector named num during the declaration. This is because the size of a vector can grow dynamically so it is not necessary to define it.

C++ Vector Initialization

```
vector<int> vector1 = {1, 2, 3, 4, 5};

                                or

vector<int> vector2 {1, 2, 3, 4, 5};

                                or

vector<int> vector3(5, 12); //initialized with size 5 and value 12.
```

Add Elements to a Vector

To add a single element into a vector, we use the push_back() function. It inserts an element into the end of the vector. For example,

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> num {1, 2, 3, 4, 5};//this may raise error

  cout << "Initial Vector: ";

  for (const int& i : num) {//this may raise error
    cout << i << "  ";
  }

  // add the integers 6 and 7 to the vector
  num.push_back(6);
  num.push_back(7);

  cout << "\nUpdated Vector: ";

  for (const int& i : num) {//this may raise error
    cout << i << "  ";
  }

  return 0;
}
```

Here, we have initialized an int vector num with the elements {1, 2, 3, 4, 5} and the push_back() function adds elements 6 and 7 to the vector.

Output:

```
Initial Vector: 1  2  3  4  5
Updated Vector: 1  2  3  4  5  6  7
```

Access Elements of a Vector

In C++, we use the index number to access the vector elements. Here, we use the at() function to access the element from the specified index. For example,

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> num {1, 2, 3, 4, 5}; // this may raise error

  cout << "Element at Index 0: " << num.at(0) << endl;
  cout << "Element at Index 2: " << num.at(2) << endl;
  cout << "Element at Index 4: " << num.at(4);

  return 0;
}
```

Output:

```
Element at Index 0: 1
Element at Index 2: 3
```

Access Elements of a Vector

We can change an element of the vector using the same at() function. For example,

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> num {1, 2, 3, 4, 5};//this may raise error

  cout << "Initial Vector: ";

  for (const int& i : num) {//this may raise error
    cout << i << "  ";
  }

  // change elements at indexes 1 and 4
  num.at(1) = 9;
  num.at(4) = 7;

  cout << "\nUpdated Vector: ";

  for (const int& i : num) {// this may raise error
    cout << i << "  ";
  }

  return 0;
}
```

Output:

```
Initial Vector: 1  2  3  4  5
Updated Vector: 1  9  3  4  7
```

Delete Elements of a Vector

To delete a single element from a vector, we use the pop_back( ) function. For example,

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> prime_numbers{2, 3, 5, 7};//this may raise error

  // initial vector
  cout << "Initial Vector: ";
  for (int i : prime_numbers) {//this may raise error
    cout << i << " ";
  }

  // remove the last element
  prime_numbers.pop_back();

  // final vector
  cout << "\nUpdated Vector: ";
  for (int i : prime_numbers) {//this may raise error
    cout << i << " ";
  }

  return 0;
}
```

Output:

```
Initial Vector: 2 3 5 7
Updated Vector: 2 3 5
```

C++ Vector Functions

In C++, the vector header file provides various functions that can be used to perform different operations on a vector.

Function	Description
size()	returns the number of elements present in the vector
clear()	removes all the elements of the vector
front()	returns the first element of the vector
back()	returns the last element of the vector
empty()	returns **1** (true) if the vector is empty
capacity()	check the overall size of a vector

We can declare an iterator for each container in the C++ Standard Template Library. For example,

```
vector<int>::iterator it;
```

We often use iterator member functions like begin( ), end( ), etc. to return iterators that point to container elements. For example,

```
vector<int> numbers = {3, 2, 5, 1, 4};
vector<int>::iterator itr1 = numbers.begin();
vector<int>::iterator itr2 = numbers.end();
```

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    // initialize vector of int type
    vector<int> numbers {1, 2, 3, 4, 5};//this may raise error

    // initialize vector iterator to point to the first element
    vector<int>::iterator itr = numbers.begin();
    cout << "First Element: " << *itr << "  "<<endl;

    // change iterator to point to the last element
    itr = numbers.end() - 1;
    cout << "Last Element: " << *itr;

    return 0;
}
```

In the above code, the begin( ) and end( ) member functions of the vector container return iterators pointing to the first and one-past-the-last elements, respectively. Here, we have used numbers.end( )- 1 instead of numbers.end( ). This is because the end( ) function points to the theoretical element that comes after the final element of the container. So, we need to subtract 1 from numbers.end( ) in order to point to the final element. Similarly, using the code numbers.end( )- 2 points to the second-last element, and so on.

The iterators are then used in a for loop to iterate over the elements, and the values are printed to the console. Iterators provide a flexible and generic way to work with elements in containers and sequences, enabling algorithms and operations to be performed uniformly across different data structures. They allow you to access and manipulate data efficiently and provide a powerful toolset for working with collections of elements.

Maps

Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have the same key values.

```
#include <iostream>
#include <map>
using namespace std;
int main( ){
  // Create a map of strings to integers
      map<string, int> m;

 // Insert some values into the map
     m["one"] = 1;
     m["two"] = 2;
     m["three"] = 3;

  // Get an iterator pointing to the first element in the map
     map<string, int>::iterator it = m.begin();

  // Iterate through the map and print the elements
      while (it != m.end( )){
           cout << "Key: " << it->first << ", Value: " << it->second
<< endl;
           ++it;
       }
  return 0;
```

```
}
```

Sets

Sets are a type of associative container in which each element has to be unique because the value of the element identifies it. The values are stored in a specific sorted order i.e. either ascending or descending. The set class is the part of C++ Standard Template Library (STL) and it is defined inside the <set> header file.

Syntax:

```
set <data_type> set_name;
```

Example

```
#include <iostream>
#include <set>
using namespace std;
int main(){
    set<char> a;
    set<char>::iterator p;
    a.insert('G');
    a.insert('F');
    a.insert('G');

    p=a.begin();
    while(p!=a.end()) {
        cout << *p << ' ';
        p++;
    }
    cout << '\n';
    return 0;
}
```

Exercise

1. Create a C++ Program to implement login. It should accept user name and password and throw a custom exception if the password has less than 6 characters or does not contain a digit.
2. Create a class for dynamic stack using vectors. Implement the push( ), pop( ) and peek( ) functions. The object of stack class will be used to take any sentence as input and it should have a method reverse( ) which will reverse each word in the string.
3. You are given N integers. Store N integers in a vector and write a function Sort for sorting the N integers and print the sorted order. Now use the sorting algorithms provided in STL for sorting the vector. Measure the time taken by the two methods for sorting the vector and print the results. [Hint: You can use built-in function time( ) or anyother built-in method for measuring the processing time of sorting operation.]
4. You are a teacher and want to keep track of your students' grades. You decide to use the map container in C++ to store the student names as keys and their corresponding grades as values. Design and implement a program in C++ to fulfill the following requirements:
   - Input: Prompt the user to enter the names and grades of multiple students. Allow the user to continue entering data until they choose to stop.
   - Storage: Store the student names as keys and their grades as values in a map container.
   - Retrieval: Implement a function to retrieve the grade associated with a given student's name. If the student is not found in the map, display an appropriate message.
   - Update: Implement a function to update the grade of a specific student. Prompt the user to enter the new grade and update the corresponding value in the map.
   - Deletion: Implement a function to remove a student and their grade from the map based on a

given student's name.

- Display: Display all the student names and their corresponding grades stored in the map.

Testing: Write a program to test the functionality of your map implementation. Add multiple students with their grades, retrieve grades for specific students, update grades, delete students, and display the final list of students and their grades. As you implement the program, consider using appropriate data types, input validation, error handling, and clear user prompts. Remember to thoroughly test your implementation to ensure correctness and accuracy in storing and retrieving the student grades.

5. You are hosting a party and want to keep track of the unique guests attending. To achieve this, you decide to use the set container in C++ to store the names of the guests. Design and implement a program in C++ to fulfill the following requirements:
   - Input: Prompt the user to enter the names of multiple guests attending the party. Allow the user to continue entering names until they choose to stop.
   - Storage: Store the guest names in a set container, which will automatically enforce uniqueness by discarding duplicate names.
   - Display: Display all the unique guest names stored in the set in alphabetical order.
   - Count: Display the total number of unique guests attending the party.

Testing: Write a program to test the functionality of your set implementation. Add multiple guest names, including duplicates, and ensure that only unique names are stored in the set. Display the final list of unique guest names and the total count of guests attending the party. As you implement the program, consider using appropriate data types, input validation, error handling, and clear user prompts.