Agenda

- STL
- File IO

map

- map is a sorted associative container that contains key-value pairs with unique keys.
- Keys are sorted by using the comparison function Compare.
- Search, removal, and insertion operations have logarithmic complexity.
- Maps are usually implemented as Red–black trees
- Iterators of map iterate in ascending order of keys, where ascending is defined by the comparison that was used for construction.
- Iterator of map are bidirectional iterator that supports dereferencing (*, ->) and bidirectional movement (++, --).

iterator

- An iterator in C++ is an object that enables traversal over the elements of a container (such as std::vector, std::list, std::map, etc.) and provides access to these elements.

- Iterators are a fundamental part of the Standard Template Library (STL) and are designed to abstract the concept of element traversal, making it possible to work with different containers in a consistent manner.

- Key Characteristics of Iterators:

1. Traversal: Iterators allow you to move through the elements of a container, one element at a time.
2. Access: Iterators provide access to the element they point to, typically through the dereference operator (*).
3. Type-Specific: Iterators are strongly typed, meaning that an iterator for an std::vector will be different from an iterator for an std::list.

Types of Iterators:

1. Input Iterators:

- Can read elements from a container. Only allow single-pass access (i.e., you can only move forward through the container).

2. Output Iterators:

- Can write elements to a container.
- Also allow only single-pass access.

3. Forward Iterators:

- Can read and write elements.
- Support multi-pass traversal, meaning you can go through the container multiple times.

4. Bidirectional Iterators:

- Can move both forward and backward in a container.
- Support all operations of forward iterators, with the additional ability to decrement the iterator.

5. Random Access Iterators:

- Provide all the capabilities of bidirectional iterators.
- Allow direct access to any element in the container using arithmetic operations like addition and subtraction.

Common Operations on Iterators:

- Dereferencing (*): Access the element the iterator points to.
- Incrementing (++): Move the iterator to the next element.
- Decrementing (--): Move the iterator to the previous element (not supported by input or output iterators).
- Equality/Inequality (==, !=): Compare iterators to check if they point to the same position.
- Addition/Subtraction (+, -): For random access iterators, allows moving the iterator by a specific number of elements.

Stream

- We give input to the executing program and the execution program gives back the output.
- The sequence of bytes given as input to the executing program and the sequence of bytes that comes as output from the executing program are called stream.
- In other words, streams are nothing but the flow of data in a sequence.
- The input and output operation between the executing program and the devices like keyboard and monitor are known as “console I/O operation”.
- The input and output operation between the executing program and files are known as “disk I/O operation”.
- The I/O system of C++ contains a set of classes which define the file handling methods
- These include ifstream, ofstream and fstream classes. These classes are derived from fstream and from the corresponding iostream class.
- These classes are designed to manage the disk files, are declared in fstream and therefore we must include this file in any program that uses files.
- Standard Stream Objects of C++ associated with console:
    1. cin -> Associated with Keyboard
    2. cout -> Associated with Monitor
    3. cerr -> Error Stream
    4. clog -> Logger Stream
- ifstearm is a derived class of istream class which is declared in std namespace. It is used to read record from file.
- ofstearm is a derived class of ostream class which is declared in std namespace. It is used to write record inside file.
- fstream is derived class of iostream class which is declared in std namespace. It is used to read/write record to/from file.

stream hirerachy

Classes for File stream operations

- ios:
  - ios stands for input output stream.
  - This class is the base class for other classes in this class hierarchy.
  - This class contains the necessary facilities that are used by all the other derived classes for input and output operations.
- istream :
  - istream stands for input stream.
  - This class is derived from the class ‘ios’.
  - This class handle input stream.
  - The extraction operator(>>) is overloaded in this class to handle input streams from files to the program execution.
  - This class declares input functions such as get(), getline() and read().
- ostream :
  - ostream stands for output stream.
  - This class is derived from the class ‘ios’.
  - This class handle output stream.
  - The insertion operator(<<) is overloaded in this class to handle output streams to files from the program execution.
  - This class declares output functions such as put() and write().
- ifstream :
  - This class provides input operations.
  - It contains open() function with default input mode.
  - Inherits the functions get(), getline(), read(), seekg() and tellg() functions from the istream.
- ofstream :
  - This class provides output operations.
  - It contains open() function with default output mode.
  - Inherits the functions put(), write(), seekp() and tellp() functions from the ostream.
- fstream :
  - This class provides support for simultaneous input and output operations.
  - Inherits all the functions from istream and ostream classes through iostream.

File Handling

- A variable is a temporary container, which is used to store record in RAM.
- A file is permanent container which is used to store record on secondry storage.
- File is operating system resource.
- Types of file:
  1. Text File

2. Binary File

1. Text File

1. Example : .txt,.doc, .docx, .rtf, .c, .cpp etc
2. We can read text file using any text editor.
3. Since it requires more processing, it is slower in performance.
4. If we want to save data in human readable format then we should create text file.

2. Binary File

1. Example : .mp3, .jpg, .obj, .class
2. We can read binary file using specific program/application.
3. Since it requires less processing, it is faster in performance.
4. It doesnt save data in human readable format.

File Modes in C++

- "w" mode
  - ios_base::out:
  - ios_base::out | ios_base::trunc
- "r" mode
  - ios_base::in
- "a" mode
  - ios_base::out | ios_base::app
  - ios_base::app
- "r+" mode
  - ios_base::in | ios_base::out
- "w+" mode
  - ios_base::in | ios_base::out | ios_base::trunc
- "a+" mode
  - ios_base::in | ios_base::out | ios_base::app
  - ios_base::in | ios_base::app:
- In case of binary use "ios_base::binary"
- In C++, files are mainly dealt by using three classes fstream, ifstream, ofstream available in fstream headerfile.
- ofstream: Stream class to write on files
- ifstream: Stream class to read from files

- fstream: Stream class to both read and write from/to files.

Serilization and DeSerilization in binary Files

- When working with string data types or other derived data types (like objects or pointers) in a class and writing or reading the data to/from a binary file, you need to handle serialization and deserialization properly.
- Directly reading or writing the object's memory representation as binary data may not work correctly for derived/user defined data types due to issues like memory layout, internal pointers, and dynamic memory allocation.
- To handle string data types (and other derived data types) correctly when reading or writing binary files, you should implement custom serialization and deserialization functions in your class.
- These functions should convert your object's data into a binary representation (serialization) and reconstruct the object from binary data (deserialization).

```
// Seralizing employee class with datamembers int id,string name,double salary.
void seralize(ofstream &fout)
{
    fout.write(reinterpret_cast<const char *>(&empid), sizeof(int));
    size_t length = name.size();
    fout.write(reinterpret_cast<const char *>(&length), sizeof(size_t));
    fout.write(name.c_str(), length);
    fout.write(reinterpret_cast<const char *>(&salary), sizeof(double));
}

//Deseralizing employee class
void deseralize(ifstream &fin)
{
  fin.read(reinterpret_cast<char *>(&empid), sizeof(int));
  size_t length;
  fin.read(reinterpret_cast<char *>(&length), sizeof(size_t));
  char *buffer = new char[length + 1];
  fin.read(buffer, length);
  buffer[length] = '\0';
  name = buffer;
  delete[] buffer;
  fin.read(reinterpret_cast<char *>(&salary), sizeof(double));
}
```