# Lab 6

# I/O Library

- File Class
  - C++ provides the following classes to perform input and output of characters to/from files

| Class | Explanation |
|---|---|
| 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 |

- These classes are derived directly or indirectly from the classes istream and ostream.
- We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream.
- Therefore, we have already been using classes that are related to our file streams.
- And in fact, we can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files.

```
1  #include <iostream>
2  #include <fstream>
3
4  using namespace std;
5
6  int main() {
7
8      ofstream myfile;
9      myfile.open("example.txt");
10     myfile << "Writing this to a file.\n";
11     myfile.close();
12     return 0;
13 }
```

# I/O Library

- Open a file
  - In order to open a file with a stream object, we use its member function open:

    open (filename, mode);

  - Where filename is a string representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

| | |
|---|---|
| **ios::in** | Open for input operations. |
| **ios::out** | Open for output operations. |
| **ios::binary** | Open in binary mode. |
| **ios::ate** | Set the initial position at the end of the file.<br>If this flag is not set, the initial position is the beginning of the file. |
| **ios::app** | All output operations are performed at the end of the file, appending the content to the current content of the file. |
| **ios::trunc** | If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one. |

- All the flags can be combined using the bitwise operator OR (|).

```
 8      ofstream myfile;
 9      myfile.open("example.bin", ios::out | ios::app | ios::binary);
10
11      //conduct the same opening operation
12      ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
13
```

- To check if a file stream was successful opening a file, you can do it by calling to member *is_open*. This member function returns a bool value of *true* in the case that indeed the stream object is associated with an open file, or *false* otherwise:

```
15      if (myfile.is_open()){
16          //blah blah
17      }
```

# I/O Library

- Closing a file
  - When we are finished with our input and output operations on a file we shall close it so that the operating system is notified and its resources become available again. For that, we call the stream's member function *close*. This member function takes flushes the associated buffers and closes the file:

    myfile.close();

# Text files

- Writing operations on text files:

```cpp
//writing on a text file
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    ofstream myfile ("example.txt");
    if(myfile.is_open())
    {
        myfile << "This is a line. \n";
        myfile << "This is another line. \n";
        myfile.close();
    }
    else cout << "Unable to open file";
    return 0;

}
```

# Text files

- Reading from a file can also be performed In the same way that we did with cout:

```
1  //reading on a text file
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5
6  using namespace std;
7
8  int main() {
9      string line;
10     ifstream myfile ("example.txt");
11     if(myfile.is_open())
12     {
13         while (getline(myfile, line))
14             cout << line << '\n';
15         myfile.close();
16     }
17     else cout << "Unable to open file";
18     return 0;
19
20 }
```

# Containers

- Containers
  - A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.
  - The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).
  - Containers replicate structures very commonly used in programming: dynamic arrays (vector), queues (queue), stacks (stack), heaps (priority_queue), linked lists (list), trees (set), associative arrays (map)...

# Map

- Containers
  - **Maps** are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order.
  - In a map, the key values are generally used to sort and uniquely identify the elements, while the mapped values store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type value_type, which is a pair type combining both:

```
typedef pair<const Key, T> value_type;
```

# Map

- Containers
  - Internally, the elements in a map are always sorted by its key following a specific strict weak ordering criterion indicated by its internal comparison object.
  - map containers are generally slower than unordered_map containers to access individual elements by their key, but they allow the direct iteration on subsets based on their order.
  - The mapped values in a map can be accessed directly by their corresponding key using the find function. (find()).
  - Maps are typically implemented as binary search trees.

# Map

```cpp
#include <iostream>
#include <map>

using namespace std;

int main() {
    map <int, int> m;

    m.insert(pair<int, int>(5, 100));
    m.insert(pair<int, int>(3, 100));

    pair <int, int> p(9, 50);
    m.insert(p);

    m[12] = 200;    //insert key/value
    m[11] = 300;
    m[13] = 40;

    map <int, int>::iterator iter;
    for(iter = m.begin(); iter != m.end(); ++iter)
        cout << "(" << (*iter).first << "," << (*iter).second << ")" << " ";
    cout<<endl;


    return 0;

}
```

# Set

- set
  - **Sets** are containers that store unique elements following a specific order.
  - In a set, the value of an element also identifies it (the value is itself the key, of type T), and each value must be unique.
  - The value of the elements in a set cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.
  - Internally, the elements in a set are always sorted following a specific strict weak ordering criterion indicated by its internal comparison object. Sets are typically implemented as binary search trees.

# Set

```
2  #include <iostream>
3  #include <map>
4
5  using namespace std;
6
7  int main() {
8      set <int> s;
9
10     s.insert(40);
11     s.insert(80);
12
13     set <int>::iterator iter;
14     for (iter = s.begin(); iter != s.end(); ++iter)
15         cout << *iter << " ";
16     cout << endl;
17
18     return 0;
19
20 }
21
```