Lesson 12: Standard Template Library (STL)
General

- STL: is a standard built-in library of C++
- Implements popular data structures and algorithms
- Constructed from function and class templates → allow working with generic data types
- Defined in a namespace named “std”

Main features:
- Basic data types: string, complex
- Input/output (IO)
- Container classes: list, vector, deque, stack, map, set,…
- Traverse elements of containers (iterators)
- Popular algorithms: searching, comparison, sorting,…
- Memory management, smart pointers
- Exception handling
String manipulation

- #include <string>
- **string** class for ASCII strings, and **wstring** Unicode ones
- Basic operations: +, += (concatenation); ==, !=, >, <, >=, <= (comparison); << (output), >> (input)
- String length: int string::length() const
- Extract sub-string:
  string string::substr(int off, int count) const
- Find sub-string:
  int string::find(const char* str, int pos) const
- Convert to C string:
  const char* string::c_str() const
- Convert to number and vice versa (C++11):
  [int|long|float|double] sto[i|l|f|d](const string& s);
  string to_string([int|long|float|double] n);
  wstring to_wstring([int|long|float|double] n);
String manipulation: Example

```cpp
string s1, s2("test123");
cin >> s1;
s1 += to_string(123);
cout << (s2==s1 ? "same" : "different") << endl;

int pos = s2.find("est");
string s3 = s2.substr(pos, 4);

char s4[100];
strcpy(s4, s3.c_str());
cout << s4 << endl;
```
Containers
Arrays: vector class

- Are dynamic arrays
- Elements can be of any type (using template): `vector<type>`
- `#include <vector>`
- Example:

  ```cpp
  int p[] = {4, 2, 6};
  vector<int> a(p, p+3); // initialization from C array
  a.push_back(1);       // add to end
  a.insert(a.begin() + 2, 3); // insert at index 2
  a.insert(a.end() - 1, 5); // insert at index 1 from end
  a[3] = 10;             // get 4th element

  vector<int>::iterator i; // forward traverse
  for (i = a.begin(); i != a.end(); i++) *i += 5;

  vector<int>::reverse_iterator j; // reverse traverse
  for (j = a.rbegin(); j != a.rend(); j++)
    cout << *j << ' ';```

```
Iterators

Container classes in STL (vector, list,…) have a corresponding iterator class used to traverse elements (in forward order)

- Each iterator instance store the location of an element
- `begin()` and `end()` functions return iterator corresponding to beginning and end locations of the container
- Operators of iterator:
  - `i++` go to next element
  - `i--` go to previous element
  - `*i` value of element

Similarly, reverse_iterator class is used to traverse in reverse order

- `rbegin()` and `rend()` functions
Linked lists: list class

- Elements can be of any type (using template): `list<type>`
- `#include <list>`
- **Traversing using iterator similarly to vector**
- **Example:**
  ```
  double p[] = {1.2, 0.7, 2.2, 3.21, 6.4};
  list<double> l(p, p+5); // initialization from C array
  l.push_back(3.4); // add to end
  l.pop_front(); // remove at beginning

  list<double>::iterator i = l.begin(); // first elem.
  *i = 4.122; // assign value
  i++; // next element
  l.insert(i, 5.0); // insert element
  l.erase(i); // remove element
  l.sort(); // sort (increasing order)
  for (i = l.begin(); i != l.end(); i++) // forward traverse
    cout << *i << ' ';
  ```
Algorithms: searching

- Find min, max elements:
  - `vector<float>::iterator p = max_element(a.begin()+2, a.end()-3);`
  - `list<string>::iterator p = min_element(l.begin(), l.end());`
- Based on comparison operators → need to define if not existing

- Find exact elements:
  - `list<float>::iterator p = find(p1, p2, 2.5f);`

- Find by criteria: need to define checking functions
  - `bool isOdd(int i) { return i%2 == 1; }
    list<int>::iterator p = find_if(p1, p2, isOdd);`

- Find and replace/remove:
  - `replace_if(p1, p2, isOdd, 10);`
  - `remove_if(p1, p2, isOdd);`
Algorithms: sorting

- Sort arrays:
  - Based on comparison operators:
    - `sort(a.begin(), a.end());`
  - Need to define “<” operator for element type
  - Use custom comparison functions:
    - ```
      bool compare(const table& a, const table& b) {
        return a.c1 < b.c1 ||
            (a.c1 == b.c1 && a.c2 < b.c2);
      }
    ```
    - `sort(a.begin(), a.end(), compare);`

- Sort lists:
  - `l.sort();`
  - `l.sort(compare);`
Input/output
General

- In STL, input and output are achieved with the concept of “data streams”
  - `#include <iostream>`
  - **Output streams**: used for data output
    - `<<` operator
    - *Base class*: `basic_istream<typename>`
  - **Input streams**: used for data input
    - `>>` operator
    - *Base class*: `basic_ostream<typename>`
  - **Input/output streams**: used for both data input and output
    - `>>` and `<<` operator
    - *Base class*: `basic_iostream<typename>`
  - To enable a class working with above classes, define `>>` and `<<` operators for it
Input/output class hierarchy

```
<table>
<thead>
<tr>
<th>ios_base</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic_ios&lt;&gt;</td>
</tr>
<tr>
<td>ios/wios</td>
</tr>
<tr>
<td>basic_ifstream&lt;&gt;</td>
</tr>
<tr>
<td>ifstream/wifstream</td>
</tr>
<tr>
<td>basic_iostream&lt;&gt;</td>
</tr>
<tr>
<td>iostream/woistream</td>
</tr>
<tr>
<td>basic_fstream&lt;&gt;</td>
</tr>
<tr>
<td>fstream/wfstream</td>
</tr>
<tr>
<td>basic_stringstream&lt;&gt;</td>
</tr>
<tr>
<td>stringstream/wstringstream</td>
</tr>
<tr>
<td>basic_ostream&lt;&gt;</td>
</tr>
<tr>
<td>ostream/wostream</td>
</tr>
<tr>
<td>basic_filebuf&lt;&gt;</td>
</tr>
<tr>
<td>filebuf/wfilebuf</td>
</tr>
<tr>
<td>basic_stringbuf&lt;&gt;</td>
</tr>
<tr>
<td>stringbuf/wstringbuf</td>
</tr>
<tr>
<td>basic_streambuf&lt;&gt;</td>
</tr>
<tr>
<td>streambuf/wstreambuf</td>
</tr>
</tbody>
</table>
```
Standard input/output

- **Objects**
  - *cin*: instance of *istream* class, corresponding to stdin
  - *cout*: instance of *ostream* class, corresponding to stdout
  - *cerr*: instance of *ostream* class, corresponding to stderr
  - *wcin, wcout, wcerr*: same functions to above, but to work with Unicode

- **Example:**
  - ```cpp
    int i;
    float a[10];
    cout << "Enter i and a[i]: ";
    if (cin >> i >> a[i]) {
        cout << "a[" << i << "] = " << a[i] << endl;
        cout.flush();
    } else cerr << "Input error" << endl;
  ```

- **Read a string:**
  - ```cpp
    string s;
    getline(cin, s);
  ```

- **Attn:** avoid mixing C and C++ I/O mechanisms
Formatting output data

- Formatting functions:
  - `setf(fmtflags flag, fmtflags mask): change formatting flags`
  - `dec/hex/oct: integer base 10/16/8 (basefield)`
  - `fixed/scientific: floating-point decimal or scientific format (floatfield)`
  - `internal/left/right: margin alignment (adjustfield)`
  - `width(int w): width of fields`
  - `precision(int p): floating-point number precision`
  - **Example:**
    
    ```cpp
    cout.width(15);
    cout.setf(ios::right, ios::adjustfield);
    cout.setf(ios::scientific, ios::floatfield);
    cout.precision(3);
    cout << 34.5678;
    ```

- Using manipulators:
  - **Including:** `internal, left, right, dec, hex, oct, fixed, scientific, setprecision(p), setw(w), setiosflag(flags), endl, ends, flush`
  - `#include <iomanip>`
  - **Example:**
    ```cpp
    cout << setw(15) << right << scientific << setprecision(3) << 34.5678;
    ```
File input/output

- `#include <fstream>`
- Using `ifstream (read only)`, `ofstream (write only)`, `fstream (read/write)`
- Read/write using `>>` and `<<` operators similarly to standard input/output
- Open a file:
  - `ifstream f1("ten file", ios::in | ios::binary);`
  - `ofstream f2; f2.open("ten file", ios::out | ios::trunc);`
- Open modes:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>app</td>
<td>Always sets pointer to end of file to write</td>
</tr>
<tr>
<td>ate</td>
<td>Sets pointer to end of file</td>
</tr>
<tr>
<td>in</td>
<td>Allows reading</td>
</tr>
<tr>
<td>trunc</td>
<td>Clears old content on open</td>
</tr>
<tr>
<td>out</td>
<td>Allows writing</td>
</tr>
<tr>
<td>binary</td>
<td>Binary file</td>
</tr>
</tbody>
</table>

- Close a file:
  - `f.close();`
- Automatically closed by destructor on object destruction
- Attn: when using `fstream` for both reading and writing: before switching from reading to writing, and vice versa, must call `seekg/seekp(...)`
Binary file input/output

- Open a file: add `ios::binary` flag

- Read data:
  - `file.read(char* buffer, int size)`
  - `file.gcount()` // number of bytes read

- Write data:
  - `file.write(char* buffer, int size)`

- Check I/O errors:
  - `file.read/write(...)`
    - `if (!file) {...}
     if (!file.read/write(...)) {...}

- Move file I/O pointers: C++ uses two distinct pointers for I/O
  - Move read pointer: `file.seekg(int pos, ios::beg/cur/end)`
  - Query read pointer location: `file.tellg()`
  - Move write pointer: `file.seekp(int pos, ios::beg/cur/end)`
  - Query write pointer location: `file.tellp()`
File input/output: example (copy a file)

```cpp
bool copy_file(const char* src, const char* dst) {
    ifstream fs(src, ios::in | ios::binary);
    ofstream fd(dst, ios::out | ios::binary
                | ios::trunc);
    if (!fs || !fd) return false;

    char buf[1024];
    while (fs) {
        fs.read(buf, sizeof(buf));
        fd.write(buf, fs.gcount());
    }
    return true;
}
```
String input/output

- #include <sstream>
- Using istreamstring (read only), ostreamstring (write only), stringstream (read/write)
- Input from string:
  - string s("10 3.56 y");
  - istreamstring str(s);
  - int i; double d; char c;
  - str >> i >> d >> c;
  - Can be used for extracting data from strings
- Output to string:
  - ostreamstring str;
  - str << "i=\" << i << \", d=\" << d << \", c=\" << c;
  - string s = str.str();
  - Can be used in formatting data to strings
Overloading >> and << operators

- Data I/O in C++ is based on overloading >> and << operators
  - `ostream& operator <<(ostream& os, char c);`
  - `ostream& operator <<(ostream& os, const char* s);`
  - `ostream& operator <<(ostream& os, double n);`
  - ...
  - `istream& operator >>((istream& is, char& c);`
  - `istream& operator >>((istream& is, const char* s);`
  - `istream& operator >>((istream& is, double& n);`
  - ...

- These operators return the ostream/istream object passed from argument to enable chaining:
  - `is >> a >> b >> c;`
  - `os << a << b << c;`

- Overloading >> and << operators for custom classes:
  - `ostream& operator <<(ostream& os, const Ellipse& e) {`
    - `return os << e.rx << e.ry;`
  }
  - `istream& operator >>((istream& is, Ellipse& e) {`
    - `return is >> e.rx >> e.ry;`
Object serialization

- Conversion of an object with arbitrary data structure into a ordered data stream to enable saving/sending objects and reconstruct them
- Used in communication and data storage
- With STL, it is possible to define >> and << operators for object serialization

Example:

```cpp
istream& operator >> (istream& is, Student& s) {
    return is >> s.name >> s.sh_year >> s.yob;
}

ostream& operator << (<< s.yob;
} ostream& os, Student& s) {
    return os << s.name << s.sh_year
}...
Problems

1. Write a program to input an array of integers with arbitrary number of elements, then print even elements by traversing the array

2. Modify Prob. 1 to use `find_if(...)` instead of traversing array

3. Modify above program to use linked list instead of array

4. Given two arrays a1 and a2 in increasing order, write function to mix them into a new array a3 also in increasing order

5. Read data from a file into a list of lines, then reprint lines with length between 10 and 20 characters

6. Define `<<` and `>>` operators for Fraction class and try to use them to read/write data with cin/cout, file, string

7. Define `<<` and `>>` operators for Complex class to read/write data in binary form