CSCI251 Advanced Programming

Generic Programming V: The Standard Template Library (STL)

UNIVERSITY OF WOLLONGONG AUSTRALIA

Outline

- STL
- Why/when should (not) you use STL?
- The building blocks.
- Examples

STL

- The Standard Template Library for C++
- Based on function and class templates
- It's an evolving standard for generic programming.
  - So making use of compile time polymorphism rather than the run time polymorphism we saw in object oriented programming.

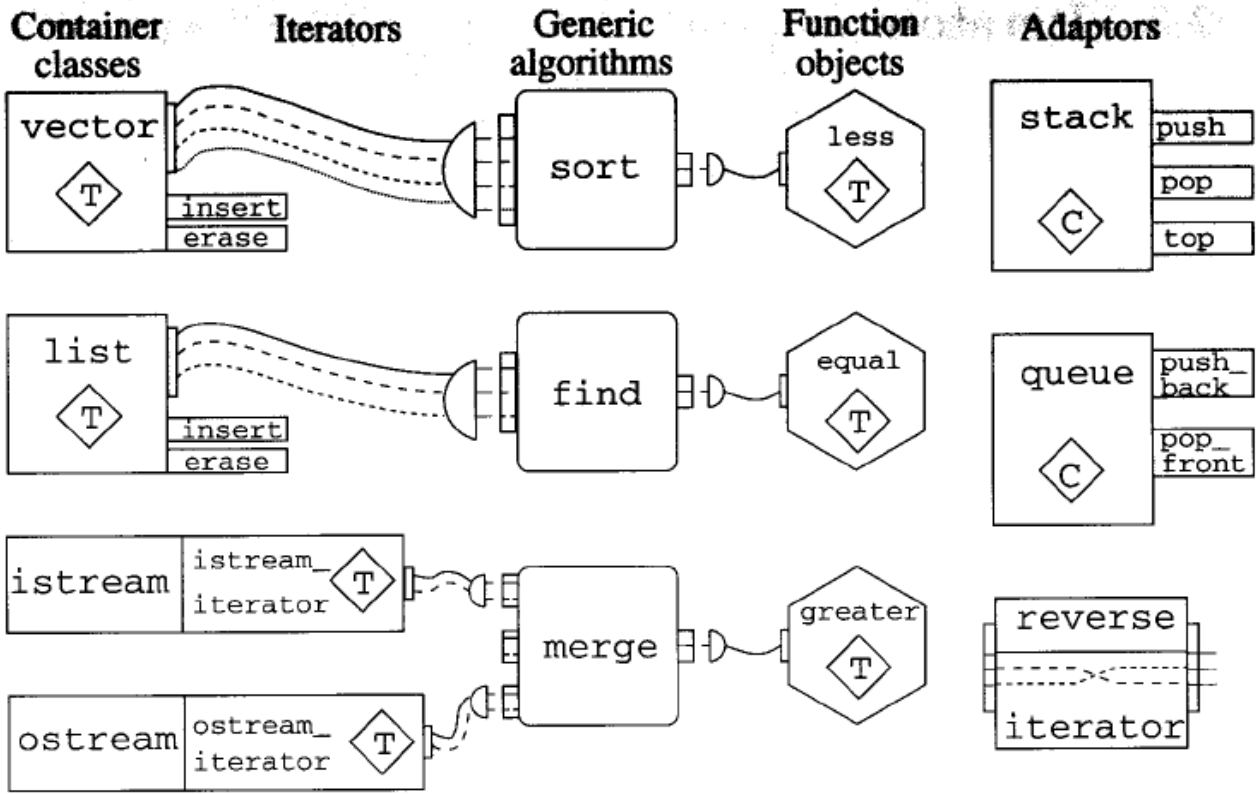
STL

- In object oriented programming we attempt to tightly bind data and the operations on the data.
  - The member variable and function
  - Potentially the data structure implies the algorithms used on it.
    - Like only human can use tools
- In generic programming we attempt to decouple the data and the operations on them.
  - The data goes into containers
  - We use the iterators as an interface to apply standard algorithms on data.

STL

- used to provide reusable, efficient and widely adaptable
- STL contains six major kinds of building blocks, implemented using templates:
  - Containers.
  - Iterators.
  - Generic algorithms.
  - Function objects.
  - Adaptors.
  - Allocators.
  - .

building blocks and relationship



Not every algorithm and iterator can be connected.

STL

- based on (function and class) templates, it can process all built in and user defined data types.
- efficiency:
  - some components generated from a template become specialized for one particular data type / scenarios
  - Very efficient for this data type
- memory size:
  - With many different types , a large number of specialized components may result in substantial size expansion.

Advantage

- process all built in and user defined data types
- reusable (as standard)
- performance guarantee
  - been written and tested well
  - efficiency
  - even with non-templated code

when should you use STL

- There is a good fit, if existing STL algorithms and containers best suit your problem.
  - Example: queuing system, dictionary
  - (although they come with different costs)
- Be compatible:
  - Make your code to follow standard conventions and be portable
- “Modern C++ programs should use the library containers rather than more primitive structures like arrays.”

Why/when shouldn’t you use STL

- Too generic may not be good
  - it will sometimes be possible to tailor things more efficiently.
  - add more functions/operations (than existing ones)
  - data structure is easier than the operation

STL Containers

- There are two types of containers:
  - Sequential containers
    - elements are organised linearly
    - position of elements is determined when they are inserted
    - array
  - Associative containers
    - elements are not necessarily organised linearly
    - elements are stored and linked on the basis of a key value
    - dictionary

Sequential containers

Container	Notes
`vector`	Flexible-size array. Supports fast random access. Inserting/deleting other than at the back may be slow.
`deque`	Double-ended queue. Supports fast random access. Fast insert/delete at front or back.
`list`	Doubly linked list. Supports only bidirectional sequential access. Fast insert/delete at any point.
`forward_list`	Singly linked list. Supports only sequential access in one direction. Fast insert/delete at any point.
`array`	Fixed-size array. Supports fast random access. Cannot add or remove elements (positions).
`string`	Specialised, not fully templated. Similar to vector but for characters. Fast random access. Fast insert/delete at the back.

random access → Access elements in an arbitrary order with similar performance.
sequential access → you need to go through elements to reach the one you want.

Sequential containers

- Default: `vector`.
- Need random access to elements: `vector` or `deque`.
- Insert or delete elements in the middle: `list` or `forward_list`.
- Insert or delete elements only at the front and back: `deque`.
- More complex scenarios:
  - Insert elements in order and subsequently needing random access.
  - Possibly just use `vector` anyway with elements to be added at the end, following by a call to `sort` it once the input is finished.

vector

- `vector<T>`: constructors
- Default, with an empty vector.

```
vector<int> v0;
```

- Initialisation with values …

```
vector<int> v1(10, -1);  // 10 ints set to -1
```

- Initialisation without values …

```
vector<int> v2(10);  // 10 ints, default set to 0.
```

- By copying from another suitable `vector<T>`.

```
vector<int> v3(v2);
vector<int> v4=v2;  // equiv. to above
```

vector

- Iterator based construction:

```
vector<int> v5(v4.begin(),v4.end());
```

- List initialisation, from C++11.

```
vector<string> words = {"one", "two", "red", "blue"};
vector<int> numbers{1,2,3,4,5,6,7};
```

vector

- The elements of a vector can be vectors themselves.
  - Vector of vectors
- A special notation was needed, the addition of a space between the last >,
  - `vector<vector<int> > v6;`

Sequential containers: Adding and removing

- Pushing and popping …

```
void push_back(const T& x);
void push_front(const T& x); // for deque
void pop_back();
void pop_front();            // for deque
```

- The use of pop removes an element but doesn't free the associated memory ➔ allocator (coming soon)
- or the use of `insert`.

examples

```
int i = 2;
// inserts 7 at i-th
index
myvector.insert(myvector
.begin() + i, 7);
```

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

int main()
{
  std::vector<int> myvector;
  int myint;

  std::cout << "Please enter integers (enter 0 to end):\n";

  do {
    std::cin >> myint;
    myvector.push_back (myint);
  } while (myint);

  std::cout << "myvector stores " << int(myvector.size()) << " numbers.\n";
  cout<<*(myvector.begin())<<endl;  // any problem
  cout<<*(--myvector.end())<<endl;  // any problem
  return 0;
}
```

examples

```
#include <array>
```

- There are a few different ways to set up an array…

```
array<int, 3> a1{ {1, 2, 3} };
array<int, 3> a2{1, 2, 3};
array<int, 3> a3 = {1, 2, 3};
array<string, 2> a4 = {string("a"), "b" };
```

examples

```
#include <iostream>
#include <array>
using namespace std;
int main()
{
    array<int, 5>a{1,2,3};
    cout << &a[2] << " " <<
   &a[0] + 2 << endl;
    return 0;
}
```

```
#include <iostream>
#include <array>
#include<string.h>
using namespace std;
int main()
{
    array<char, 50>a{1,2,3};
    strcpy(&a[0],
   "http://c.biancheng.net/stl");
    cout<<a[0]<<"+++"<<a[1]<<endl;
    cout<<a.data()<<endl;

    array<char, 50> url1{"uow.edu.au"};
    array<char, 50> url2{"nsw.gov.au"};
    url1.swap(url2);
    cout<<url1.data();
    return 0;
}
```

https://en.cppreference.com/w/cpp/container/array

Associative containers

- Sorted: Elements ordered by key:

Container	Notes
`map`	AKA Associative array; holds key-value pairs.
`set`	Container in which the key is the value.
`multimap`	`map` but with a key that can appear multiple times.
`multiset`	`set` but with a key that can appear multiple times.

Associative containers

- Unordered Collections

Container	Notes
`unordered_map`	`map` organised by a hash function.
`unordered_set`	`set` organised by a hash function.
`unordered_multimap`	Hashed `map`; keys can appear multiple times.
`unordered_multiset`	Hashed `set`; keys can appear multiple times.

Associative containers

- Across the 8 types, the 3 different parameters are:
  - Set or map, in the former the key *is* the value while in the latter there are key-value pairs.
  - Unique keys or multiple keys.
  - Ordered or not.

```
set<string> exclude = {“the”, “but”, “and”, “or”, “an”, “a”};

map<string, string> authors = {
{“A-last”, “A-first”}, {“B-last”, “B-first”}, {key, value} };
```

Set

```
#include <iostream>
#include <set>
#include <map>
using namespace std;
int main()
{
    set<int> exclude = {1, 1, 2};   // exclude.insert(1);
    if (exclude.count(1)!=0)
      std::cout << "1 is an element of the set.\n";
      cout<<exclude.size();
      for (auto i =exclude.begin(); i!= exclude.end();i++)
        cout<<*i<<endl;
    return 0;
}
```

Map

- Elements within a map are `pairs`.
- A `pair` takes two type names, and the data elements of the `pair` have the corresponding types.
  - For example,

```
pair<string, string> writers;
pair<string, string> musician{“Billy”, “Joel”};
```

- To access the data members: Use `first`, `second`.

```
musician.first (->);
musician.second;
```

Map

```
#include <map>
#include <string>
#include <iostream>
using namespace std;
int main()
{
    map<int, string> mapStudent;
    mapStudent.insert(pair<int, string>(1, "student_one"));
    mapStudent.insert(pair<int, string>(2, "student_two"));
    mapStudent.insert(pair<int, string>(3, "student_three"));
    map<int, string>::iterator iter;
    for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)
    {
        cout<<iter->first<<"  "<<iter->second<<endl;
    }
    cout<<mapStudent.size();
}
```

```
mapStudent[1] = “student_one”;
mapStudent[2] = “student_two”;
mapStudent[3] = “student_three”;
```

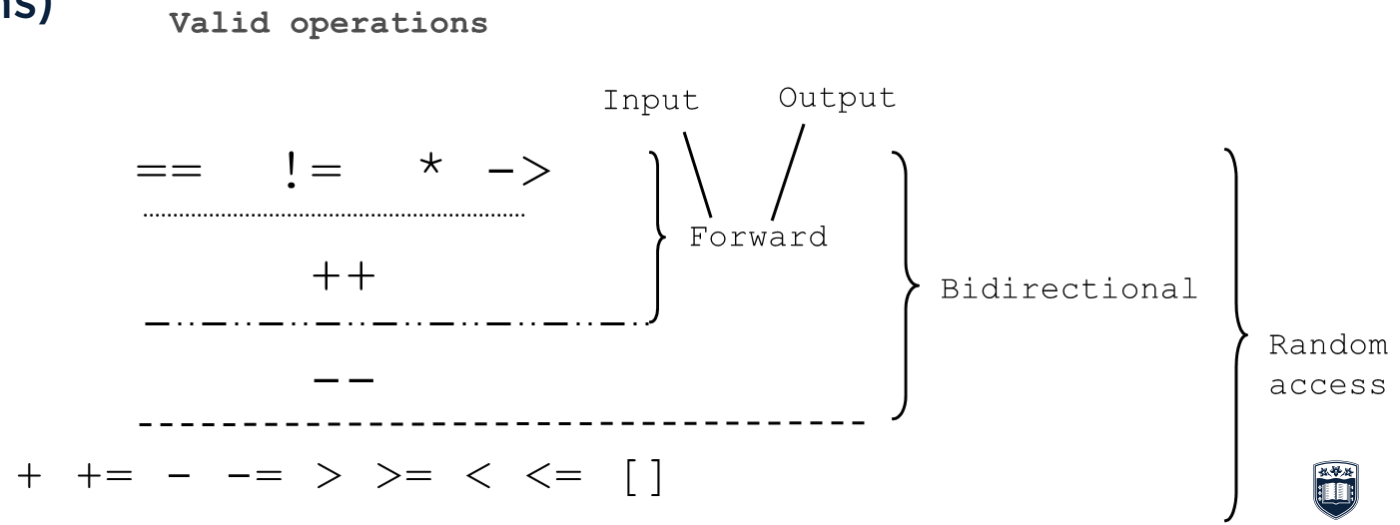
Map

```
map<int, string>::iterator iter;
iter = mapStudent.find(1);
if(iter != mapStudent.end())
{
    Cout<<"Find, the value is "<<iter->second<<endl;
}
else
{
    Cout<<"Do not Find"<<endl;
}
```

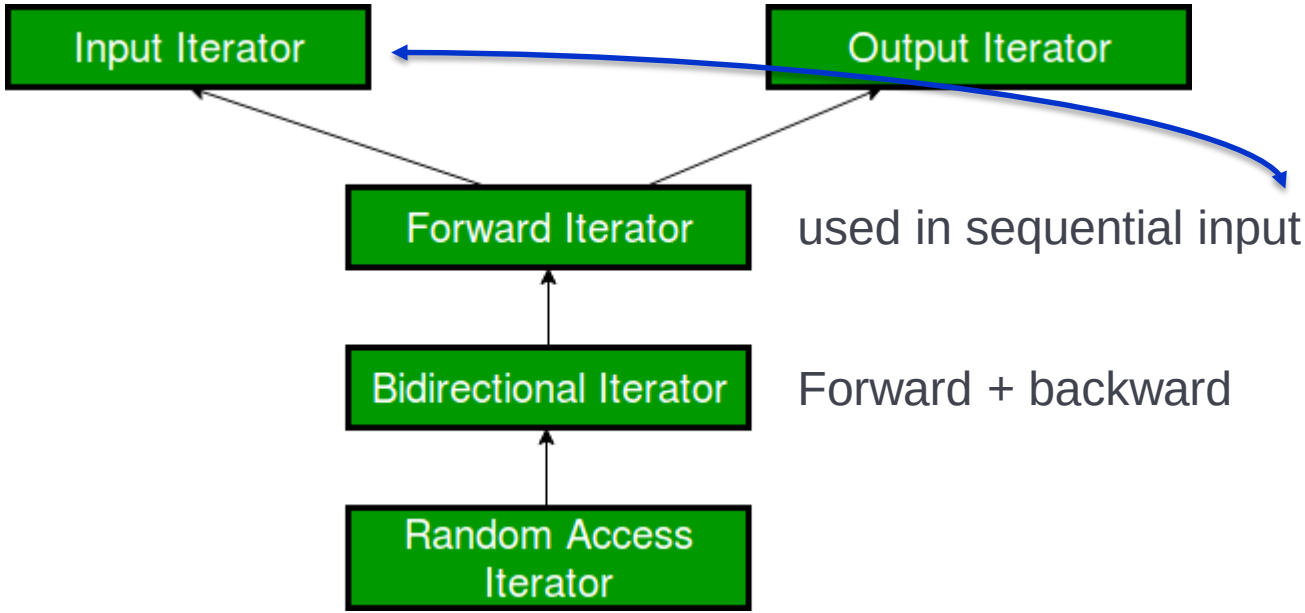
```
if (mapStudent.count(1)!=0)
    {std::cout << "1 is an element of the set.\n";
    cout<<mapStudent[1]<<endl;
    }
```

Iterators

- different categories of iterator (for particular generic algorithms)



Iterators



used in sequential input operations, read only

Forward + backward

https://www.geeksforgeeks.org/input-iterators-in-cpp/

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

int main()
{
  vector<char> c;
  c.push_back('A');
  c.push_back('B');
  c.push_back('C');
  c.push_back('D');
  for (int i=0; i<4;++i)
    cout << "c[“ << I << "]=“ << c[i] << endl;
  vector<char>::iterator p = c.begin();
  cout << "The third entry is “ << c[2] << endl;
  cout << "The third entry is “ << p[2] << endl;
  cout << "The third entry is “ << *(p+2) << endl;

  cout << "Back to c[0].\n";
  p = c.begin();
  cout << "which has value “ << *p << endl;
```

```
 cout << "Two steps forward and
one step back:\n";

  p++;
  cout << *p << endl;
  p++;
  cout << *p << endl;
  p--;
  cout << *p << endl;
  return 0;
}
```

```
c[0]=A
c[1]=B
c[2]=C
c[3]=D
The third entry is C
The third entry is C
The third entry is C
Back to c[0].
which has value A
Two steps forward and one step back:
B
C
B
```

Reverse iterator

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

int main()
{
  vector<char> c;
  c.push_back('A');
  c.push_back('B');
  c.push_back('C');
  cout << "Forward:\n";
  vector<char>::iterator p;

  for (p=c.begin(); p!=c.end();
p++) cout<< *p << " ";
  cout << endl;
```

```
  cout << "Reverse:\n";
  vector<char>::reverse_iterator
rp;

  for(rp=c.rbegin(); rp!=c.rend();
rp++) cout<< *rp <<" ";
  cout << endl;

  return 0;
}
```