Container

容器

A personal notebook

- It allows notes to be stored.
- It has no limit on the number of notes it can store.
- It will show individual notes.
- It will tell us how many notes it is currently storing.
- Array can not be used here.

Collection

- Collection objects are objects that can store an arbitrary number of other objects.

What is STL

- STL = Standard Template Library
- Part of the ISO Standard C++ Library
- Data Structures and algorithms for C++

Why should I use STL?

- Reduce development time.
  - Data-structures already written and debugged.
- Code readability
  - Fit more meaningful stuff on one page.
- Robustness
  - STL data structures grow automatically.
- Portable code.
- Maintainable code
- Easy

C++ Standard Library

Library includes:

- A Pair class (pairs of anything, int/int, int/char, etc)
- Containers
  - vector (expandable array)
  - deque (expandable array, expands at both ends)
  - list (double-linked)
  - sets and maps
- Basic Algorithms (sort, search, etc)
- All identifiers in library are in the `std` namespace: `using namespace std;`

The ‘Top 3’ data structures

- `map`
  - Any key type, any value type.
  - Sorted.
- `vector`
  - Like C array, but auto-extending.
- `list`
  - doubly-linked list.

All Sequential Containers

- `vector` : variable array
- `deque` : dual-end queue
- `list` : double-linked-list
- `forward_list` : as it
- `array` : as “array”
- `string` : char. array

Example using the vector class

```
#include <iostream>
#include <vector>

using namespace std;

int main( ) {
    //  Declare a vector of ints (no need to worry about size)
    vector<int> x;
    //  Add elements
    for (int a=0; a<1000; a++)
        x.push_back(a);
    //  Have a pre-defined iterator for vector class, can use it to print out the items in vector
    vector<int>::iterator p;
    for (p=x.begin(); p<x.end(); p++)
        cout << *p << " ";
    return 0;
}
```

Generic Classes

```
vector<int> x;
vector<string> notes;
```

- Have to specify two types:
    i. the type of the collection itself (here: `vector`) and
    ii. the type of the elements that we plan to store in the collection (here: `string`)

vector

- It is able to increase its internal capacity as required: as more items are added, it simply makes enough room for them.
- It keeps its own private count of how many items it is currently storing. Its size method returns the number of objects currently stored in it.
- It maintains the order of items you insert into it. You can later retrieve them in the same order.

Basic Vector Operations

- Constructors

```
vector<Elem> c;
```

```
vector<Elem> c1(c2);
```

- Simple Methods

```
V.size( ) // num items
```

```
V.empty( ) // empty?
```

```
==, !=, <, >, <=, >=
```

```
V.swap(v2) // swap
```

- Iterators

```
I.begin( ) // first position
```

```
I.end( ) // last position
```

- Element access

  `V.at(index)`

  `V[index]`

  `V.front() // first item`

  `V.back( ) // last item`

- Add/Remove/Find

  `V.push_back(e)`

  `V.pop_back( )`

  `v.insert(pos, e)`

  `V.erase(pos)`

  `V.clear( )`

  `V.find(first, last, item)`

Two ways to use Vector

- Preallocate

```
vector<int> v(100);
v[80]=1; // okay
v[200]=1; // bad
```

- Grow tail

```
vector<int> v2;
int i;
while (cin >> i)
    v.push_back(i);
```

List Class

- Same basic concepts as vector
  - Constructors
  - Ability to compare lists (`==` , `!=` , `<` , `<=` , `>` , `>=`)
  - Ability to access front and back of list

```
x.front(), x.back()
```

- Ability to assign items to a list, remove items

```
x.push_back(item)
x.push_front(item)
x.pop_back()
x.pop_front()
x.erase(pos1, pos2)
```

Sample List Application

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

int main( ) {
    //  Declare a list of strings
    list<string> s;
    s.push_back("hello");
    s.push_back("world");
    s.push_front("tide");
    s.push_front("crimson");
    s.push_front("alabama");
    list<string>::iterator p;
    for (p=s.begin(); p!=s.end(); p++)
       cout << *p << " ";
    cout << endl;
}
```

What's wrong with the following code:

```
list<int> lst1;
list<int>::iterator iter1 = lst1.begin();
list<int>::iterator iter2 = lst1.end();
while (iter1 < iter2) {
    //...
}
```

Example of List

```
list<int> L;
for(int i=1; i<=5; ++i)
    L.push_back(i);
//delete second item.
L.erase( ++L.begin() );
copy( L.begin(), L.end(),ostream_iterator<int>(cout, ",")); // Prints: 1,3,4,5,
cout << endl;
```

Maintaining an ordered list

```
#include <iostream>
#include <list>
#include <string>
using namespace std;
int main( ) {
    list<string> s;
    string t;
    list<string>::iterator p;
    for (int a=0; a<5; a++) {
        cout << "enter a string : ";
        cin >> t;
        p = s.begin();
        while (p != s.end() && *p < t)
            p++;
        s.insert(p, t);
    }
    for (p=s.begin(); p!=s.end(); p++)
        cout << *p << " ";
    cout << endl;
}
```

Choose Between Sequential Containers

- Use `vector` unless you have other reasons
- Don't use `list` or `forward_list` if your program has lots of small elements and space overhead matters
- Use `vector` or `deque` if the program requires random access to elements
- Use `list` or `forward_list` if the program needs to insert elements in the middle of the container
- Use `deque` if the program needs to insert elements at the front and the back, but not in the middle

Maps

- 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 mapped values in a map can be accessed directly by their corresponding key using the bracket operator (operator[]).
- Maps are typically implemented as binary search trees.

Example Map Program

```
#include <map>
#include <string>

map<string,float> price;
price["snapple"] = 0.75;
price["coke"] = 0.50;
string item;
double total=0;
while ( cin >> item )
    total += price[item];
```

Simple Example of Map

```
map<long,int> root;
root[4] = 2;
root[1000000] = 1000;
long l;
cin >> l;
if (root.count(l))
    cout<<root[l]
else cout<<"Not perfect square";
```

Example

```
// Create a map of three (string, int) pairs
std::map<std::string, int> m{{"CPU", 10}, {"GPU", 15}, {"RAM", 20}};

print_map("1) Initial map: ", m);

m["CPU"] = 25; // update an existing value
m["SSD"] = 30; // insert a new value
print_map("2) Updated map: ", m);

// Using operator[] with non-existent key always performs an insert
std::cout << "3) m[UPS] = " << m["UPS"] << '\n';
print_map("4) Updated map: ", m);

m.erase("GPU");
print_map("5) After erase: ", m);

m.clear();
std::cout << std::boolalpha << "6) Map is empty: " << m.empty() << '\n';
```

Iterator

Iterators

- Declaring

```
list<int>::iterator li;
```

- Front of container

```
list<int> L;
li = L.begin();
```

- Past the end

```
li = L.end();
```

Iterators

- Can increment

```
list<int>::iterator li;
list<int> L;
li=L.begin();
++li; // Second thing;
```

- Can be dereferenced

```
*li = 10;
```

Algorithms

- Take iterators as arguments

```
list<int> L;
vector<int> V;
//  #include <algorithm> for this
// put list in vector
copy( L.begin(),
    L.end(),
    V.begin()
);
```

List Example Again

```
list<int> L;
for(int i=1; i<=5; ++i) {
    L.push_back(i); //delete second item.
}
L.erase(++L.begin());
copy(L.begin(), L.end(),
    ostream_iterator<int>(cout, ",")); // Prints: 1,3,4,5
cout << endl;
```

for-each loop (C++11)

- A for-each loop iterates over the elements of arrays, vectors, or any other data sets. It assigns the value of the current element to the variable iterator declared inside the loop

```
for(type variable_name : array/vector_name)
{
    loop statements
    ...
}
```

Example of for-each

```
#include<iostream>
using namespace std;
int main()
{
    int arr[]={1,2,3,4,5};   //array initialization
    cout<<"The elements are: ";
    for(int i : arr)
    {
        cout<<i<<" ";
    }
    return 0;
}
```

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<int> vec={11,22,33,44,55,66};
    cout<<"The elements are: ";
    for(auto var : vec)
    {
        cout<<var<<" ";
        }
    return 0;
}
```

Iteration of a Map

- The iteration over a `map` gets a pair of the key and the value

```
map<string, string> entries;
for (auto entry : entries)
{
    dates.push_back(entry.first + ":" + entry.second);
}
```

- The `auto` specifies the type of `entry` to be infereced from `entries`'s type at compile time

Pros and Cons

- Advantages of foreach loop
  - It eliminates the possibility of errors and makes the code more readable.
  - Easy to implement
  - Does not require pre-initialization of the iterator
- Disadvantages of foreach loop
  - Cannot directly access the corresponding element indices
  - Cannot traverse the elements in reverse order
  - It doesn’t allow the user to skip any element as it traverses over each one of them

Typdef

- Annoying to type long names
  - `map<string, list<string>> phonebook;`
  - `map<string, list<string>>::iterator finger;`
- Simplify with typedef
  - `typedef map<string,list<string>> PB;`
  - `PB phonebook;`
  - `PB::iterator finger;`
- Or use the `using` statement:
  - `using PB = map<string,list<string>>;`

Pitfalls

- Accessing an invalid

```
vector<int> v;
v[100]=1; // Whoops!
```

- Solutions:
  - use `push_back()`
  - Preallocate with constructor.
  - Reallocate with `reserve()`
  - Check `capacity()`

Pitfalls

- Inadvertently inserting into `map<>`.

```
if (foo["bob"]==1)
//silently created entry “bob”
```

- Use `count()` to check for a key without creating a new entry.

```
if ( foo.count("bob") )
```

Pitfalls

- Do not use `count()` on `list<>`

```
if ( my_list.count() == 0 ) { ... } // Slow
if ( my_list.empty() ) {...}        // Fast
```

Pitfalls

- Use invalid iterator

```
list<int> L;
list<int>::iterator li;
li = L.begin();
L.erase(li);
++li; // WRONG
// Use return value of erase to advance
li = L.erase(li); // RIGHT
```

What we've learned today?

- Referennce
- STL Containers
  - vector
  - list
  - map
- STL Iterator