Container

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.

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 std namespace

using namespace std;

The three parts of STL

- Containers
- Algorithms
- Iterators

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

- Use “namespace std” so that you can refer to vectors in C++ library
- Just declare a vector of ints (no need to worry about size)
- Add elements
- Have a pre-defined iterator for vector class, can use it to print out the items in vector

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

int main( ) {
        vector<int> x;
        for (int a=0; a<1000; a++)
            x.push_back(a);
    vector<int>::iterator p;
    for (p=x.begin();
                p<x.end(); p++)
        cout << *p << " ";
    return 0;
}
```

generic classes

```
vector<string> notes;
```

- Have to specify two types: the type of the collection itself (here: vector) and 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.

Class Exercises

- The code for the vector example exists at vector.cpp. Modify this code so it puts 5000 items in the vector, and then prints out every fifth element
  - Element 0, element 5, element 10, etc.

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)

Class Exercises

- Take a look at the code in vector2.cpp . Predict the output of this program.
- Run the program to check your output.

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.remove(item)

Sample List Application

- Declare a list of strings
- Add elements
  - –Some to the back
  - –Some to the front
- Iterate through the list
  - –Note the termination condition for our iterator

    p != s.end( )
  - –Cannot use **p < s.end( )** as with vectors, as the list elements may not be stored in order

```
#include <iostream>
using namespace std;
#include <list>
#include <string>
int main( ) {
    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;
}
```

Maintaining an ordered list

- Declare a list
- Read in five strings, add them in order
- Print out the ordered list

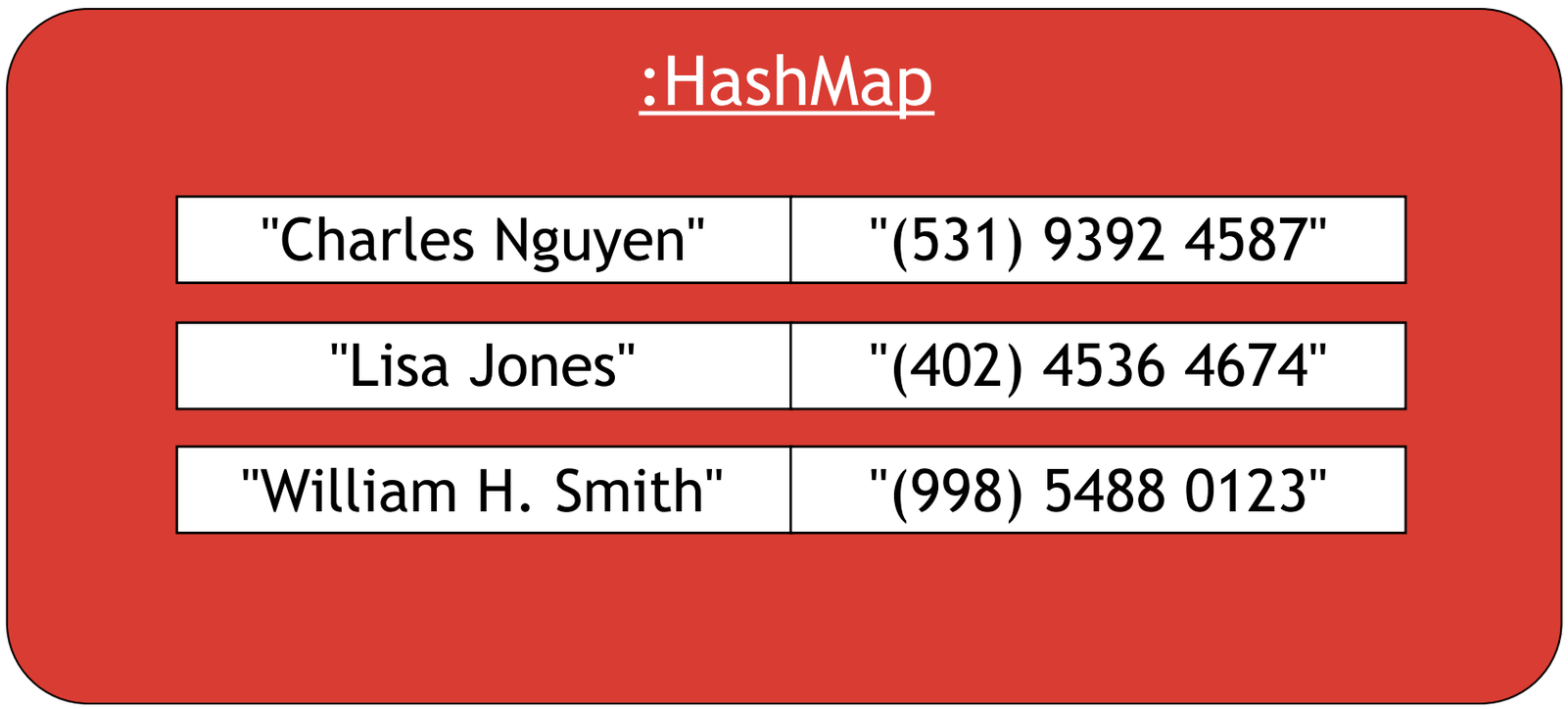
```
#include <iostream>
    using namespace std;
#include <list>
#include <string>
int main( ) {
    list<string> s;  string t;
    list<string>::iterator p;
    for (int a=0; a<5; a++) {
          cout << "enter a string : ";
          cin >> t;
          p = s.begin();
          while (p != s.end()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

Maps

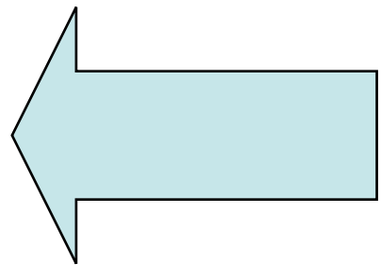
- Maps are collections that contain pairs of values.
- Pairs consist of a key and a value.
- Lookup works by supplying a key, and retrieving a value.
- An example: a telephone book.

Using maps

- A map with strings as keys and values



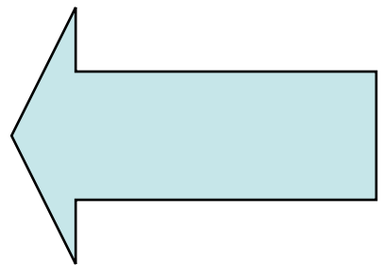
Example Program



```
#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];
```

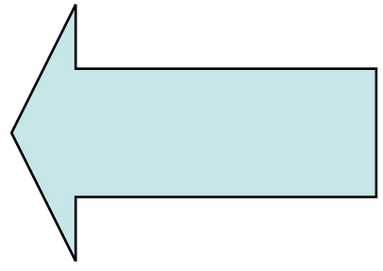
Example Program

```
#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];
```



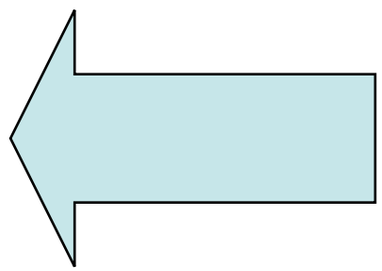
Example Program

```
#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];
```



Example 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”;
```

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);
```

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: 1,2,3,5
```

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;
// 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: 1,2,3,5
```

Typdefs

- Annoying to type long names
  - map<Name, list<PhoneNum> > phonebook;
  - map<Name, list<PhoneNum> >::iterator finger;
- Simplify with typedef
  - typedef PB map<Name,list<PhoneNum> >;
  - PB phonebook;
  - PB::iterator finger;
- Easy to change implementation.

Using your own classes in STL Containers

- Might need:
  - Assignment Operator, operator=()
  - Default Constructor
- For sorted types, like map<>
  - Need less-than operator: operator<()
    - Some types have this by default:
      - int, char, string
    - Some do not:
      - char *

Example of User-Defined Type

```
struct point
    {
    float x;
    float y;
    }
vector<point> points;
point p; p.x=1; p.y=1;
points.push_back(1);
```

Example of User-Defined Type

- Sorted container needs sort function.

```
struct full_name {
    char * first;
    char * last;
    bool operator<(full_name & a)
      {return strcmp(first, a.first) < 0;}
    }
map<full_name,int> phonebook;
```

Performance

- Personal experience 1:
  - –STL implementation was 40% slower than hand-optimized version.
    - STL: used deque
    - Hand Coded: Used “circular buffer” array;
  - –Spent several days debugging the hand-coded version.
  - –In my case, not worth it.
  - –Still have prototype: way to debug fast version.

Performance

- Personal experience 2
- Application with STL list ~5% slower than custom list.
- Custom list “intrusive”
  - struct foo {
  - int a;
  - foo * next;
  - };
- Can only put foo in one list at a time ☹

Pitfalls

- Accessing an invalid vector<> element.

```
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

- Not using empty() on list<> .
  - –Slow

```
if ( my_list.count() == 0 ) { ... }
```

  - –Fast

```
if ( my_list.empty() ) {...}
```

Pitfalls

- Using 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
```

Common Compiler Errors

- vector<vector<int>> vv;

  missing space

  lexer thinks it is a right-shift.

- any error message with pair<...>

  map<a,b> implemented with pair<a,b>

Other data structures

- set, multiset, multimap
- queue, priority_queue
- stack , deque
- slist, bitset, valarray