C++ standard classes

STL (Standard Template Library)

Examples of string construction

```
#include <string>

// by char string
char *ptr = new char[22];
ptr = "say goodnight, Gracie";
string Str1(ptr);        // create a string initialized by char* string

// or even
string Str2("say goodnight, Gracie");

// by another string
string Str3(Str2);

// by a substring of another string
string Str4(Str3, 4, 9);  // initialized to "goodnight"---a 9 character
                          // substring starting with the 4th character

// or explicit call to substr method
string Str5 = Str3.substr(4, 9) ;
```

You can find where a substring starts, erase it, and insert another

```
#include <string>
// search for string "Gracie" starting at character number 0
string::size_type pos = Str2.find("Gracie",0);
if( pos != string::npos ) {   // check that found
   Str2.erase(pos, 6);
   Str2.insert(pos, "Irene");
   cout << Str2 << endl;
}

// find the first instance of one of a number of characters
string match = "Ie";
pos = Str2.find_first_of(match,0);
cout << "Found character " << Str2[pos] << " at position " << pos << endl;
```

Appending strings can be done using '+'

```
Str3 = Str2+Str4;
```

There are times when you want a string to be represented as a char *, for use in older Unix utilities, e.g., to convert a string to an integer or floating point.

```
#include <string>
#include <stdlib.h>
string Str6("1234");
string Str7("7893.34");

// atoi and atof require char* arguments. method c_str() provides this
int   str6_int   = atoi(Str6.c_str());
float str7_float = atof(Str7.c_str());
```

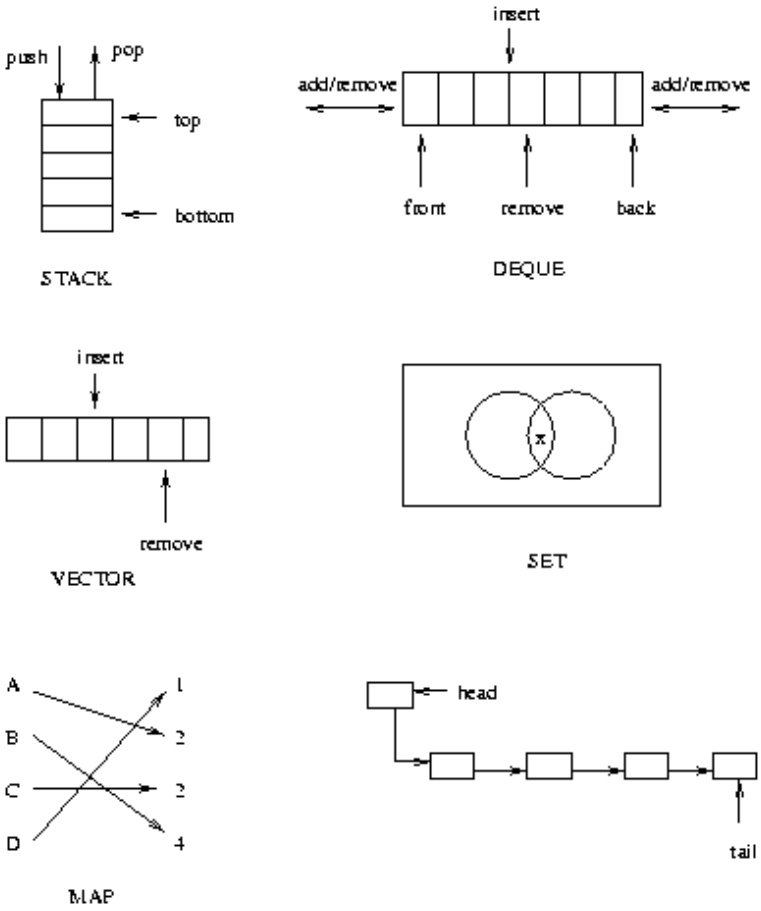
This functionality gives you almost everything you'll need with strings.

http://www.sgi.com/tech/stl/

gives a full story, but to make sense of you need to know about the other classes.

Standard C++ Classes

- string class - lots of string manipulation methods
- Containers - objects to contain groups of other objects
    - ♦ Sequence containers - `vector`, `deque`, `list`, `queue`, `stack`
    - ♦ Associative containers - `set`, `map`



Standard C++ Libraries are implemented using templates

- Declare the type of the objects to be put in the containers

```
#include <iostream>          // needed for IO
#include <string>            // needed for strings
#include <list>              // needed for list container
#include <vector>            // needed for vector container
#include <deque>             // needed for deque container
#include <queue>             // needed for queue containers
#include <set>               // needed for set container
#include <map>               // needed for map container
#include "TwoDimPt.h"        // user defined class (Lecture 1)

vector<int> vecInt;
deque<TwoDimPt<int> > dequeOfMine;
set<int> setInt;
map<string,TwoDimPt<int> > strMap;
```

- Hidden container methods assume that what ever class is being contained, has
    - ♦ default constructor, copy constructor, assignment operator (often defaults are fine)

- ♦ < operator defined on pairs of class instance
- ♦ = = operator defined on pairs of class instance

This allows the container methods to organize the objects *without knowing the type!!!*

- All interactions with a container class are through **copying**-the container class takes care of all internal memory management details.
- Some methods are common to ALL containers. Important ones
  - ♦ `int size()` // how many elements present
  - ♦ `int max_size()` // largest number that can be held
  - ♦ `bool empty()` // is the container empty?

See this example for an overview of everthing we've seen on C++ up to this point. It contains examples of operator overloading and more.

- string class - lots of string manipulation methods
- Containers - objects to contain groups of other objects
  - ♦ Sequence containers - `vector`, `deque`, `list`, `queue`, `stack`
  - ♦ Associative containers - `set`, `map`

Standard C++ Libraries are implemented using templates

- Declare the type of the objects to be put in the containers

```
#include <iostream>          // needed for IO
#include <string>            // needed for strings
#include <list>              // needed for list container
#include <vector>            // needed for vector container
#include <deque>             // needed for deque container
#include <queue>             // needed for queue containers
#include <set>               // needed for set container
nclude <map>               // needed for map container
#include "TwoDimPt.h"        // user defined class (Lecture 1)

vector<int> vecInt;
deque<TwoDimPt<int> > dequeOfMine;
set<int> setInt;
map<string,TwoDimPt<int> > strMap;
```

- Hidden container methods assume that what ever class is being contained, has
  - ♦ default constructor, copy constructor, assignment operator (often defaults are fine)
  - ♦ < operator defined on pairs of class instance
  - ♦ = = operator defined on pairs of class instance

  This allows the container methods to organize the objects *without knowing the type!!!*

  Here there is an example.

  Here there is another example.
- All interactions with a container class are through **copying**-the container class takes care of all internal memory management details.
- Some methods are common to ALL containers. Important ones
  - ♦ `int size()` // how many elements present
  - ♦ `int max_size()` // largest number that can be held
  - ♦ `bool empty()` // is the container empty?

Iterators

- Access to container elements is normally done through *Iterators*, which are generalized pointers.
  - ♦ An iterator is a class, defined in terms of the container
  - ♦ EVERY container has methods that return iterators associated with the conta iner

- ◊ `begin()`, points to 1st object
- ◊ `end()`, points to past-end-value
- ◊ `rbegin()`, points to 1st object (reverse)
- ◊ `rend()`, points to past-end-value (reverse)

♦ Iterators have associated operators '*', '++', '-'

• Consider an example which illustrates inserting, deleting, and iterating over the most basic container, the list.

```
#include <iostream>
#include <list>

void main(int argc, char* argv[]) {

 list<int> ExampleList;      // declare a list

 list<int>::iterator P;      // iterator tailored to list class. Reflects
                             // a class definition within a class (iterator
                             // class within list<class T> template)

 P = ExampleList.begin();

 // insert elements
 ExampleList.insert(P, 1);  // insert '1' at point BEFORE iterator
 ExampleList.insert(P, 2);  // insert '2' at point BEFORE iterator,
                            // which places it AFTER the previous insert
 ExampleList.insert( ExampleList.begin(), 3);  // where will it go??


 cout << "Print the inserted elements " << endl;
 P = ExampleList.begin();
 while( P != ExampleList.end() ) {

      // note the dereferencing operator on the iterator
      cout << "List element [" << *P << "]" << endl;
      P++;        // note definition of ++
 }

 // we can insert ranges of elements also. First make a copy
 list<int> AnotherExample = ExampleList;

 // now insert all but the first element of ExampleList at
 // the BACK of AnotherExample.  We need to identify the range
 // using iterators
 list<int>::iterator Q = ExampleList.begin();
 Q++;

 P = AnotherExample.end();

 // the insertion specifies (a) the point in front of which
 // the insertion will occur, (b) the starting point of the
 // the range of elements to insert, (c) the point AFTER
 // the last element to insert
 //
 AnotherExample.insert(P, Q, ExampleList.end());

 cout << "Print the inserted elements, round 2 " << endl;
 P = AnotherExample.begin();
 while( P != AnotherExample.end() ) {
      cout << "List element [" << *P << "]" << endl;
      P++;        // note definition of ++
 }

 cout << "Print the inserted elements, backwards for fun " << endl;

 // different iterator is needed
 list<int>::reverse_iterator RP = AnotherExample.rbegin();
 while( RP != AnotherExample.rend() ) {
```

```
      cout << "List element [" << *RP << "]" << endl;
      RP++;        // note definition of ++
 }

 // show how to empty a container
 AnotherExample.clear();

 cout << "Try to print the inserted elements, but find nothing " <<
endl;
 P = AnotherExample.begin();
 while( P != AnotherExample.end() ) {
      cout << "List element [" << *P << "]" << endl;
      P++;        // note definition of ++
 }
}

[lecture4]$ g++ example1.cc
[lecture4]$ ./a.out
Print the inserted elements
List element [3]
List element [1]
List element [2]
Print the inserted elements, round 2
List element [3]
List element [1]
List element [2]
List element [1]
List element [2]
Print the inserted elements, backwards for fun
List element [2]
List element [1]
List element [2]
List element [1]
List element [3]
Try to print the inserted elements, but find nothing
```

- Aside-think about
  - ♦ why the iterator class is ``co-designed" with the class it iterates over
  - ♦ why a *different* iterator class is needed to iterate backwards.
- The Vector Container
  - ♦ You can index into a queue just as you do an array because the [] operator is defined for it. Must be within current "range" of vector
  - ♦ You can dynamically resize the queue
  - ♦ index of elements may change as elements are added and deleted from the vector.
  - ♦ Example

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

void vec_status(vector<int>& V) {

cout << "size = " << V.size() << ", "
     << "capacity = " << V.capacity() << ", "
     << "max size = " << V.max_size() << ", "
     << "empty = " << V.empty() << endl;

if(! V.empty() ) {
        cout << "[";
        for(int i=0; i< V.size(); i++)
              cout << " " << V[i] ;
        cout << " ]" << endl;
}}


void main() {

vector<int> vec;
```

```
cout << "\n--display initial state--" << endl;
vec_status(vec);    // initial empty state

// allocate some space
vec.reserve(6);

cout << "\n--after reserving space for 6 elements--" << endl;
vec_status(vec);


// some action
vec.push_back(1);
vec.push_back(2);
vec.push_back(3);

cout << "\n--after pushing 1,2,3 at back--" << endl;
vec_status(vec);    // now see, note size != capacity

// insert 2 elements in
vector<int>::iterator P = vec.begin();
P += 2;           // note += is defined on iterator

vec.insert(P,4);  // plunk

cout << "\n--after a 4 inserted in front of the 3rd element--" << en
dl;
vec_status(vec);    // now see

// what happens if we do direct assignment
vec[4] = -4;        // outside of existing range
vec[3] = -3;        // within existing range

cout << "\n--after vec[4] = -4 and vec[3] = -3 --" << endl;
vec_status(vec);

cout << "\n--after 2 pop_back() calls--" << endl;
vec.pop_back();
vec.pop_back();
vec_status(vec);
}

[lecture4]$ ./a.out
--display initial state--
size = 0, capacity = 0, max size = 1073741823, empty = 1

--after reserving space for 6 elements--
size = 0, capacity = 6, max size = 1073741823, empty = 1

--after pushing 1,2,3 at back--
size = 3, capacity = 6, max size = 1073741823, empty = 0
[ 1 2 3 ]

--after a 4 inserted in front of the 3rd element--
size = 4, capacity = 6, max size = 1073741823, empty = 0
[ 1 2 4 3 ]

--after vec[4] = -4 and vec[3] = -3 --
size = 4, capacity = 6, max size = 1073741823, empty = 0
[ 1 2 4 -3 ]

--after 2 pop_back() calls--
size = 2, capacity = 6, max size = 1073741823, empty = 0
[ 1 2 ]
```

- Note that
  - `reserve` increases the capacity of the vector
  - `size` gives the number of elements in the vector

- ◊ `iterator-based insertion` ``moves'' the elements within the vector
- ◊ assignment within capacity, outside of range, doesn't "take", but within range, does
- ◊ NO ARRAY BOUNDS CHECKING IS BEING DONE!!!!

- Associative Maps - general indexing.
  - ♦ In the course project we will want to take an, say, Entity table name (e.g., Address), which formally is a string, and find a pointer to the table object associated with that name. Observe

```
#include <map>
#include <string>
#include "RDB.h"        // hypothetical database definition file
                        // RDB = Relational DataBase

map<string,RDBtable *> table_map;

string EntityName("Address");
RDBtable* RDBtptr;

...

// associate the name and the pointer
table_map[EntityName] = RDBtptr;

...
// make sure the desired association is made, get back out the pointer
if( table_map.count(EntityName) > 0 )    // count # hits
{
    RDBtptr = table_map[EntityName];     // get out the pointer
}
```

  - ♦ Observations
    - ◊ Natural indexing style
    - ◊ Bad Things Will Happen if you attempt to index out using a key that isn't present
  - ♦ A map is like any other container, and can be manipulated with iterators and insert/delete methods
  - ♦ The objects stored in a map container are a class of type `pair`

```
template<class K,class T>
class pair {
  public:
    K first;
    T second;
};
```

  - ♦ Example

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

void main(int argc, char* argv  []) {

map<string,int> mapper;

mapper[string("Three")] = 3;
mapper[string("Two")] = 2;
mapper[string("One")] = 1;

pair<string,int> object;

map<string,int>::iterator p = mapper.begin();
while( p != mapper.end() ) {
        object = *p;      // iterator points to a pair
        cout << "mapper[" << object.first << "] = " << o
bject.second << endl;
        p++;
 }
}
```

```
[lecture4]$ !g+
g++ example3.cc
[lecture4]$ ./a.out
mapper[One] = 1
mapper[Three] = 3
mapper[Two] = 2
```

- Points to note
  - ◊ sizing is automatic
  - ◊ the object held in the container is a ``pair''
  - ◊ the pairs are stored in sorted order w.r.t. the key

Using Sets

The ``set'' is another Standard C++ Library construct. To use it, put

```
#include <algorithm>
#include <set>
```

in your program. The first declaration make available some generic functions one can use to manipulate sets.

Sets are easiest to use when the objects in the set are built-in data types, e.g., int, double. strings are easy also.

Let's look at code that declares sets, and creates new sets from existing ones. The first example uses sets whose components are strings.

```
#include <iostream>
#include <string>
#include <set>
#include <algorithm>

void main() {

 // make the elements that will go into the sets
 string Str1("string1");
 string Str2("string2");
 string Str3("string3");

 // make set variables
 set<string> A;
 set<string> B;
 set<string> C;

 // use the insert operator to
 A.insert(Str1);
 A.insert(Str2);
 B.insert(Str2);
 B.insert(Str3);

 set<string>::iterator P = A.begin();
 while( P != A.end() ) {
        cout << "Set A has element " << *P << endl;
        P++;
 }

 cout << endl;

 P = B.begin();
 while( P != B.end() ) {
        cout << "Set B has element " << *P << endl;
        P++;
 }

 cout << endl;
```

```
// Make a set that is the union of two sets
set_union( A.begin(), A.end(),
           B.begin(), B.end(),
           inserter( C, C.begin() ));

P = C.begin();
while( P != C.end() ) {
       cout << "Set A union B has element " << *P << endl;
       P++;
}

// empty the set variable
C.clear();

cout << endl;

// Make a set that is the intersection of two sets
set_intersection( A.begin(), A.end(),
           B.begin(), B.end(),
           inserter( C, C.begin() ));

P = C.begin();
while( P != C.end() ) {
       cout << "Set A interset B has element " << *P << endl;
       P++;
}

// empty the set variable
C.clear();

cout << endl;

// Make a set that has elements in A which are not in B
set_difference( A.begin(), A.end(),
           B.begin(), B.end(),
           inserter( C, C.begin() ));

P = C.begin();
while( P != C.end() ) {
       cout << "Set A-B has element " << *P << endl;
       P++;
}

// empty the set variable
C.clear();

cout << endl;

// Make a set that has elements in B which are not in A
set_difference( B.begin(), B.end(),
           A.begin(), A.end(),
           inserter( C, C.begin() ));

P = C.begin();
while( P != C.end() ) {
       cout << "Set B-A has element " << *P << endl;
       P++;
}

// now eliminate Str1 from A
A.erase(Str1);

cout << endl;

P = A.begin();
while( P != A.end() ) {
       cout << "Set A (after erasing String1) has element " << *P << endl;
       P++;
```

```
 }
}
```

When we run it, we get what the expected result

```
[lecture6]$ a.out
Set A has element string1
Set A has element string2

Set B has element string2
Set B has element string3

Set A union B has element string1
Set A union B has element string2
Set A union B has element string3

Set A interset B has element string2

Set A-B has element string1

Set B-A has element string3

Set A (after erasing String1) has element string2
```

We can also define sets on user-defined objects. For this we must define operators = = and < on the object class. Let's now repeat the exercise on sets of objects comprised of a pair-an integer and a string.

```
#include <iostream>
#include <string>
#include <set>
#include <algorithm>


// the object we'll make sets of
class coord {
 private:
   int    _n;
   string _s;
 public:
   coord() {};
   coord(int n, string& s) : _n(n), _s(s) {};

   // equality if the components are equal
   bool operator==(const coord& c) {
      return ((_n == c._n) && (_s == c._s));
   }

   // first component has highest priority in < ordering
   bool operator<(const coord& c) {
     if( _n < c._n ) return true;
     if( c._n < _n ) return false;
     if( _s < c._s ) return true;
     return false;
   }

   // not absolutely necessary, just useful for this example
   friend ostream& operator<<(ostream&, coord&);
};

ostream&
operator<<( ostream& os, coord& c) {
   os << "[" << c._n << "," << c._s << "]";
}

void main() {
```

```
string Str1("string1");
string Str2("string2");
string Str3("string3");

coord c1(1, Str1);
coord c2(2, Str2);
coord c3(3, Str3);

set<coord> A;
set<coord> B;
set<coord> C;

// From here the code is almost identical to the earlier set example
A.insert(c1);
A.insert(c2);
B.insert(c2);
B.insert(c3);

set<coord>::iterator P = A.begin();
while( P != A.end() ) {
       cout << "Set A has element " << *P << endl;
       P++;
}

cout << endl;

P = B.begin();
while( P != B.end() ) {
       cout << "Set B has element " << *P << endl;
       P++;
}

cout << endl;

// Make a set that is the union of two sets
set_union( A.begin(), A.end(),
           B.begin(), B.end(),
           inserter( C, C.begin() ));

P = C.begin();
while( P != C.end() ) {
       cout << "Set A union B has element " << *P << endl;
       P++;
}

// empty the set variable
C.clear();

cout << endl;

// Make a set that is the intersection of two sets
set_intersection( A.begin(), A.end(),
           B.begin(), B.end(),
           inserter( C, C.begin() ));

P = C.begin();
while( P != C.end() ) {
       cout << "Set A interset B has element " << *P << endl;
       P++;
}

// empty the set variable
C.clear();

cout << endl;

// Make a set that has elements in A which are not in B
set_difference( A.begin(), A.end(),
```

```
            B.begin(), B.end(),
            inserter( C, C.begin() ));

 P = C.begin();
 while( P != C.end() ) {
        cout << "Set A-B has element " << *P << endl;
        P++;
 }

 // empty the set variable
 C.clear();

 cout << endl;

 // Make a set that has elements in B which are not in A
 set_difference( B.begin(), B.end(),
            A.begin(), A.end(),
            inserter( C, C.begin() ));

 P = C.begin();
 while( P != C.end() ) {
        cout << "Set B-A has element " << *P << endl;
        P++;
 }

 // now eliminate c1 from A
 A.erase(c1);

 cout << endl;

 P = A.begin();
 while( P != A.end() ) {
        cout << "Set A (after erasing c1) has element " << *P << endl;
        P++;
 }

}
```

When we compile this we get a load of warnings involving const& that I've not been able yet to figure out. The code does compile. When we run this we get

```
[nicol@ghost lecture5]$ a.out
Set A has element [1,string1]
Set A has element [2,string2]

Set B has element [2,string2]
Set B has element [3,string3]

Set A union B has element [1,string1]
Set A union B has element [2,string2]
Set A union B has element [3,string3]

Set A interset B has element [2,string2]

Set A-B has element [1,string1]

Set B-A has element [3,string3]

Set A (after erasing c1) has element [2,string2]
```