CS 294-73
Software Engineering for Scientific Computing

Lecture 7: STL Containers

Function and Operator Overloading

- We are using `+,*,-,/ ...` with many different types of arguments, different meanings in different contexts.
  - Familiar in all programming languages: a*b is understood for a,b integers, floats, …
- C++ takes this to the limit consistent with static strong typing.
  - Operators: binary infix `(x,*,...)` , prefix / postfix `(++,--)` , operator precedence is tied to the operator, but otherwise we can define them anyway we want.
  - Function names can also be overloaded
  - Uniquely determined by types of arguments, return values, classes.
- One more level of overloading can be obtained by using namespaces.

Examples of Overloading

- ```
  ostream& operator<<(ostream&,const T&)
  cout << t << ... ;
  ```

- ```
  void RectMDArray<T,N>::operator*=(const T&);
  void RectMDArray<T,N>::operator*=(const RectMDArray<T,N>);
  ```

- ```
  const T& RectMDArray<T,N>::operator()(const Point&, int) const;
        T& RectMDArray<T,N>::operator()(const Point&, int);
  ```

- Member function names. In fact, you *want* to use the same member function names for analogous functionality across multiple classes (see later with iterators).
  ```

Standard Template Library containers.

Predefined classes: aggregates that are templated on the type being held.

Example of a namespace. The names of these classes are `std::className.`

We use the command

```
using namespace std;
```

in global scope to tell compiler to look for functions of the form `std::className.` Some authorities view this as bad form.

http://www.cplusplus.com/

NB: C++11 standard.

Various choices in container templates

Container templates in the STL

- C arrays as first-class objects (array),
- dynamic arrays (vector),
- queues (queue),
- stacks (stack),
- heaps (priority_queue),
- linked lists (list),
- trees (set),
- associative arrays (map)

- They are distinguished by the kinds of access they provide and the complexity of their operations.
- To use these, you need to include the appropriate header file, e.g.
```
#include <array>
```

`Array<T,N>, pair<T1,T2>`

- Why not `int foo[3],` rather than `Array<int, 3> foo ?`
  - `array<int, 3>` is a type – objects of this type can be returned, assigned, etc.
  - `array<int,3> tupleFunction(...) // perfectly ok.`
  - `int foo[3] tupleFunction(...) // doesn't make sense.`

- `pair:` lots of circumstances where you need to hand around pairs of objects of different classes.
  - `pair<T1,T2> pr = make_pair(t1,t2);`
  - `pr.first`
  - `pr.second`

vector<T>

```
vector<int> foo;
for (int k = 0; k < 10; k++)
   {
     foo.push_back(k);
   }
for (auto it=foo.begin();it != foo.end(); ++it)
   {
     cout << *it << endl;
   }
```

`vector<T>`

Several new things:

- Classes declared inside of classes. What things can be declared inside of a class A ?
  - Functions `void A::bar(...)`
  - Data `a.m_foo` (one per object); `A::s_bar` (static, one per class).
  - Classes: `A::Aprime`;

- `vector<T>::iterator` is a class member of `vector<T>` . Abstracts the idea of location in a linearly-ordered set.
  - `it = vec.begin();` Calls a member function of vector<T> that returns an object of class `vector<T>::iterator`, initialized to initial location in `vec` .
  - `it.end() == true` if you have reached the end of vec.
  - `++it, --it` increments, decrements the location by one.
  - `*it` returns a reference to the contents at the current location in `vec`.
  - You could have gotten the same functionality by an ordinary loop and indexing, but only for vector, not for the other containers.

`vector<T>`

- `auto`
  - `(vector<T>::iterator it = vec.begin(); ...` is a lot of keystrokes.
  - `auto <varname> = ...;` can be used instead of a type declaration if the type can be inferred unambiguously from the right-hand side at compile time. In this case, `vector<T>::begin()` has not been overloaded, i.e. there is only one member function with that name and no arguments, and its return type is `vector<T>::iterator` .
  - `auto` can be used for many other things than this. For readability and self-documentation, it is probably best not to overuse it (Compilers can find meaningful interpretations of what may be typographical errors).

Adding, deleting, accessing elements of vector

```
unsigned int size();

push_back(const T&);

pop_back(const T&);

T& back();

T& front();

operator[ ](int);

Vector<T>::iterator begin()
```

• Looks like a 1D array: can index any element by an integer less than `size()` .

• Can add, delete elements at the end of an array.

• Fast access: data stored in contiguous localtions in memory (just as if you had used `new`. In fact, you can access the underlying contiguous storage as an ordinary 1D array.

Back to `vector<T>`

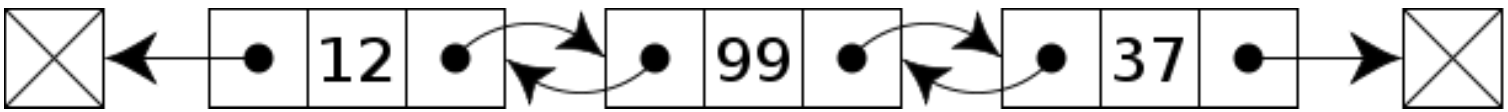
```
vector<int> foo;
for (int k = 0; k < 10; k++)
    {
      foo.push_back(k);
    }
for (auto it=foo.begin();it != foo.end(); ++it)
    {
      cout << *it << endl;
    }
```

How do remove an element from a vector ?

- Can do this at the end easily, but in general
  - find the element you wish to remove
  - make a whole new vector 1 smaller than the original
  - copy all but the excluded object to the new vector
- But we have already been doing something almost as awful with the push_back function of vector
  - grow vector length by one
  - copy all elements to the new vector with length+=1
  - copy the new element on the end
  - (in reality vector is doing a version of doubling it's size when it runs of of room and keeps track of it's "real" size and it's `size()`)
- Vectors are good at:
  - Accessing individual elements by their position index (constant time).
  - Iterating over the elements in any order (linear time).
  - Add and remove elements from its end (constant amortized time).

list<T>

- std::list provides the following features
  - Efficient insertion and removal of elements anywhere in the container (constant time).
  - Efficient moving elements and block of elements within the container or even between different containers (constant time).
  - Iterating over the elements in forward or reverse order (linear time).



- What list is not good at is random access.
  - ie. if you wanted to access the 35th entry in a list, you need to walk down the linked list to the 35th entry and return it.

list<T>

```
unsigned int size();

push_back(const T&);

pop_back(const T&);

T& front();

T& back();

insert(list<T>::iterator ,const T&);

erase(list<T>::iterator );

list<T>::iterator begin();
```

But no indexing operator ! However, insertion / deletion is cheap once you find the location you want to insert or delete at.

Why list instead of vector ?

- `erase, insert, splice, merge` are O(1) complexity
- `remove, unique` are O(linear) complexity.

```
void removeBoundary(std::list<Node>& a_nodes, std::list<Node>&
  a_boundary)
{
   std::list<Node>::iterator it;
   for(it=a_nodes.begin(); it!=a_nodes.end(); ++it)
    {
     if(!it->isInterior())
     {
       a_boundary.splice(a_boundary.start(), a_nodes, it);
     }
}
```

Executes in linear time, and Node is never copied.

<map> : an associative container

- Stores elements formed by the combination of a *key value* and a *mapped value*.
- You index into a map with the key, you get back out the value.
  - You could consider vector a simple map where the key is an unsigned integer, and the value is the template class, but that imposes the constraint that they keys are the continuous interval [0,size-1]
  - but what if your keys don't have this nice simple property ?
- map take two template arguments, one for the key, and one for the value
- The key class needs to implement the operator<
  - Strict Weak Ordering
    - if a<b and b<c, then a<c
    - if a<b then !(b<a)
    - if !(a<b) and !(b<a) then a == b

<map> : an associative container

- Stores elements formed by the combination of a *key value* and a *mapped value*.
- You index into a map with the key, you get back out the value.
  - You could consider vector a simple map where the key is an unsigned integer, and the value is the template class, but that imposes the constraint that they keys are the continuous interval [0,size-1]
  - but what if your keys don't have this nice simple property ?
- map take two template arguments, one for the key, and one for the value
- The key class needs to implement the operator<
  - Strict Weak Ordering
    - if a<b and b<c, then a<c
    - if a<b then !(b<a)
    - if !(a<b) and !(b<a) then a == b

Map<Key,T>

```
unsigned int size();

insert(pair<Key,T>);
insert(list<T>::iterator ,const T&);

erase(list<T>::iterator );
erase(const Key K&);

T& front();
T& back();

list<T>::iterator begin();
operator[](const Key K&);
```

A simple dictionary object

```
#include <map>
#include <string>
#include <iostream>
using namespace std;
void fillDictionary(map<string,string>& a_dictionary, const string& filename);
int main(int argc, char* argv[])
{
  map<string, string> dictionary;
  string key;
  fillDictionary(dictionary, argv[1]);
  while(true)
    {
      cout<<"query :";
      cin>>key;
      if(key.size()==0) return 0;
      map<string,string>::iterator val = dictionary.find(key);
      if(val==dictionary.end())
        cout<<"\n did not find that word in the dictionary "<<endl;
      else
        cout<<"\n"<<val->second<<endl;
    }
}
```

parse a simple input file

```
void fillDictionary(map<string,string>& a_dictionary, const string&
  filename)
{
  ifstream f(filename.c_str()); string key, value; char buffer[2048];
  bool next=true; char token[] =":";
  while(f.getline(buffer,2048, token[0]))
    {
      if(next)
       {
         key = string(buffer);  next = !next;
       }
      else
      {
        value = string(buffer);  a_dictionary[key]=value; next=!next;
      }
    }
}
```

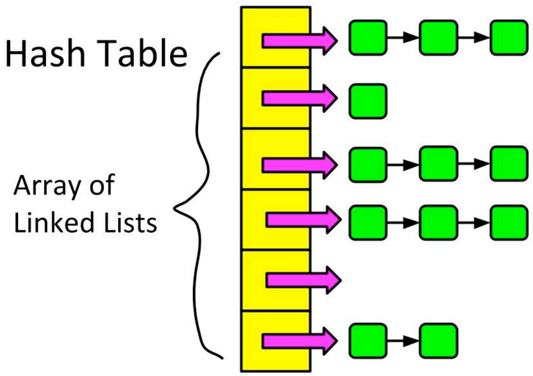
Access with operator[ ]
keys are unique
if not found, makes new entry

std::unordered_map

- Same Interface as std::map, but this is a *hash table*
- Optimized for fast lookup
  - std::map is O(logN) insertion and lookup, std::unordered_map is O(logN) insertion and O(1) lookup
  - The constant is non-trivial
- Implementation generally uses more memory to speed up lookup (one or two levels of binning)
- Relies on the concept of *hashing*
  - Turn Key type into a size_t integer.

```
std::size_t h = std::hash<Bob>(myBob);
```

  - A good hash has few if any collisions
  - A good container hash even density



- Encryption hashing has different goals
  - Nearby Key types should hash to very different integers
  - you use more bits to have lots of room and reduce the probability of collision
- Not a good choice if your access pattern is visiting every member

Hierarchical use of `RectMDArray`

```
/// Boolean-valued RectMDArray. Defines which Boxes are
members of the domain.

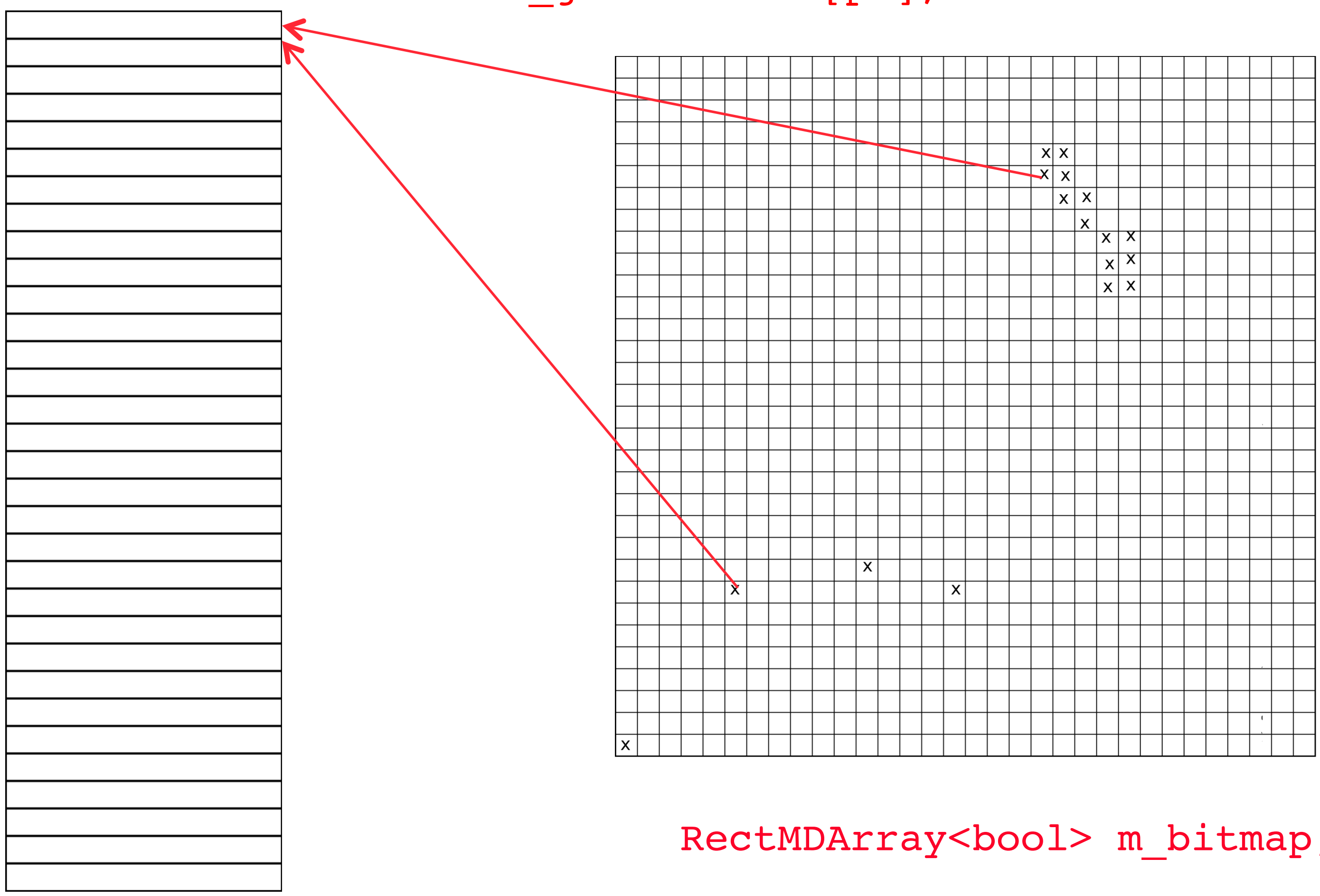
RectMDArray<bool> m_bitmap;

/// Vector of Points each of which is associated with a
data in the region defined by a Point: a fefined patch of
grid, a collection of particles.
vector<T> m_stuff;

/// Maps Points to an index into m_stuff.
map<Point, int > m_getPatches;
```

Hierarchical use of RectMDArray

```
map<Point,int> m_getContents;
int k = m_getContents[pt];
```



```
RectMDArray<bool> m_bitmap;
```

```
vector<list<Particle > > m_particles;
vector<RectMDArray<T,N> > m_refGrids;
```