CS32 Spring 2021

Week 6 Dis 1F

TA: Manoj Reddy

LA: Katherine Zhang

Credit: Prof. Nachenberg

Outline

- Templates/STL

Generic Programming

- Goal: To build algorithms that can operate on many different types of data (not just a single type).
- Examples:
  - Sort function to sort ints, strings, objects etc.
  - Linked List class that hold ints, strings, objects etc.
- A generic function or class can be quickly reused to solve many different problems
- Improves efficiency (faster programming)

Part 1: Generic Comparisons

- Compare 2 objects of same class like 2 integers
- Unlike an assignment operator, only it compares two objects instead of assigning one to another
  - You can define ==, <, >, <=, >= and !=
- Can only use public members when defined outside the class
- getWeight() needs to be const function
- Note: ‘const’ keyword

```
bool operator<(const Dog &other)
const
{
  if (m_weight < other.m_weight)
     return true;
  return false; // otherwise
}
```

```
bool operator>=(const Dog &a, const
Dog &b)
{
  if (a.getWeight() >= b.getWeight())
    return(true);
  else return(false);
}
```

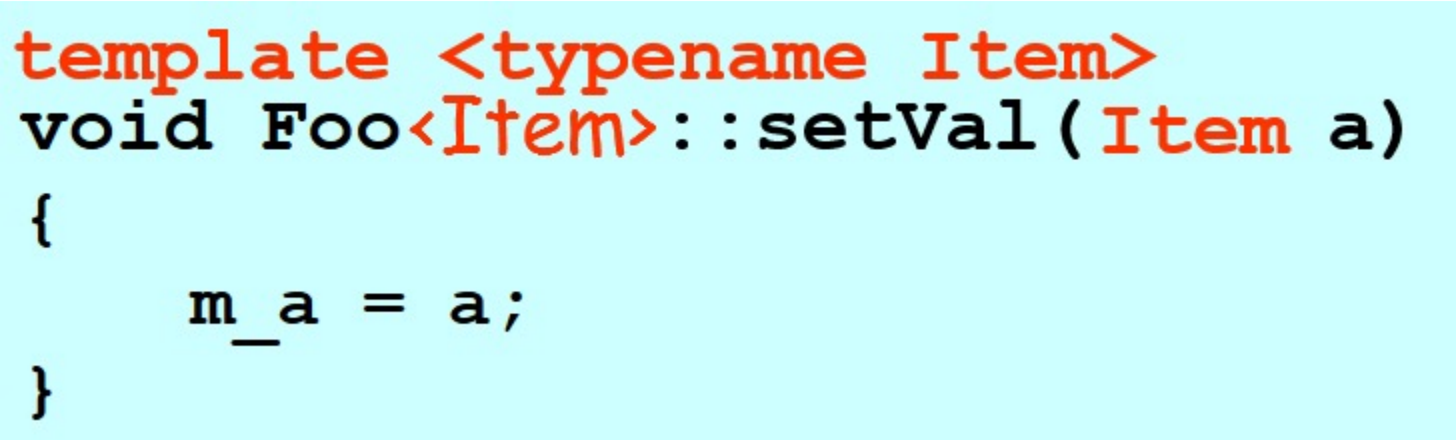
Part 2: Generic Functions

- Functions that can take generic parameter types
- Compiler generates a new version of the function for every type
  - Cons: Increases the size of the compiled program
  - Pros: Time-saving, Bug-reducing, Source-simplifying
- Must use template type to define at least one formal parameter
- Don’t assume comparison operators are defined
- Multi-type templates:
  - template <typename Type1, typename Type2>
- Can be over-ridden using specialized implementation

```
template <typename Item>
void swap(Item  &a, Item  &b)
{
 Item  temp;
 temp = a;
 a = b;
 b = temp;
}

int main()
{
  Dog d1(10), d2(20);
  Circle c1(1), c2(5);

  swap(d1,d2);
  swap(c1,c2);
  …
}
```

Part 3: Generic Classes

- Instead of functions, classes can also contain template types
- Same syntax as before
- Example:
  - stack<int>, stack<string>
  - queue<int>, queue<string>



```
template <typename Item>
void Foo<Item>::setVal(Item a)
{
    m_a = a;
}
```

Part 4: Standard Template Library (STL)

- The Standard Template Library or STL is a collection of pre-written, tested classes provided by the authors of C++.
- These classes were all built using templates, meaning they can be used with many different data types.
- Examples:
  - **stack**
  - **queue**
  - vector
  - list
  - map
  - set

Vector

- *vector<int> vals(2,444);*
- Works exactly like an array, only it doesn’t have a fixed size
- Vectors grow/shrink automatically when you add/remove items
- Can access elements using indices ([])
- Insert
  - push_back inserts an item to the end of the vector
    - May grow in size
- Remove
  - pop_back removes an item from the back of a vector
  - May shrink in size
- Useful functions
  - empty(), size()

List

- list<float> lf;
- Works exactly like a linked list
- Like vector, the list class has push_back, pop_back, front, back, size and empty methods!
- But it also has push_front and pop_front methods!
- Unlike vectors, you can’t access list elements using brackets.
- Random access:
  - Using iterators

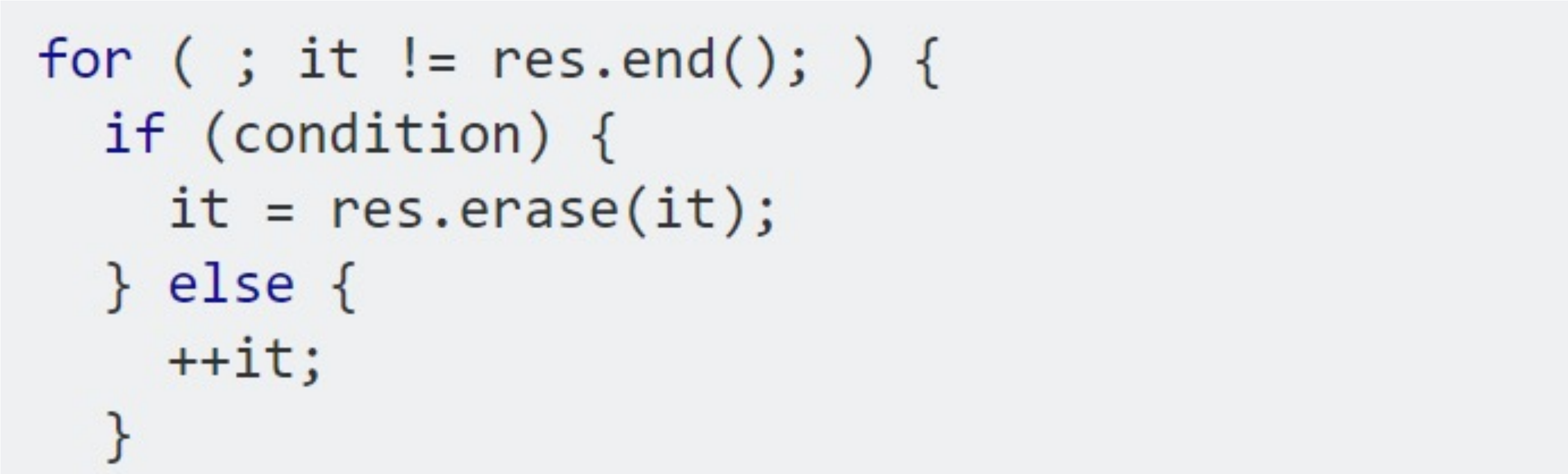
Iterators

- Iterator variable is just like a pointer variable, but it’s used just with STL containers.
- Can move iterator down one item by using by ++ operator (-- operator to move backward)
- Works with structs/classes

```
list <Car> cars;
Car toyota;
cars.push_back(toyota);
list<Car>::iterator it = cars.begin();
(*it).getCarName();
it->getCarName();
```

```
main()
{
 vector<int>       myVec;

   myVec.push_back(1234);
   myVec.push_back(5);
   myVec.push_back(7);

     vector<int>::iterator  it;

     it = myVec.begin();

     while (  it != myVec.end()  )
     {
        cout << (*it);
        it++;
     }
}
```

Deletion using Iterators



```
for ( ; it != res.end(); ) {
  if (condition) {
    it = res.erase(it);
  } else {
    ++it;
  }
```

Map

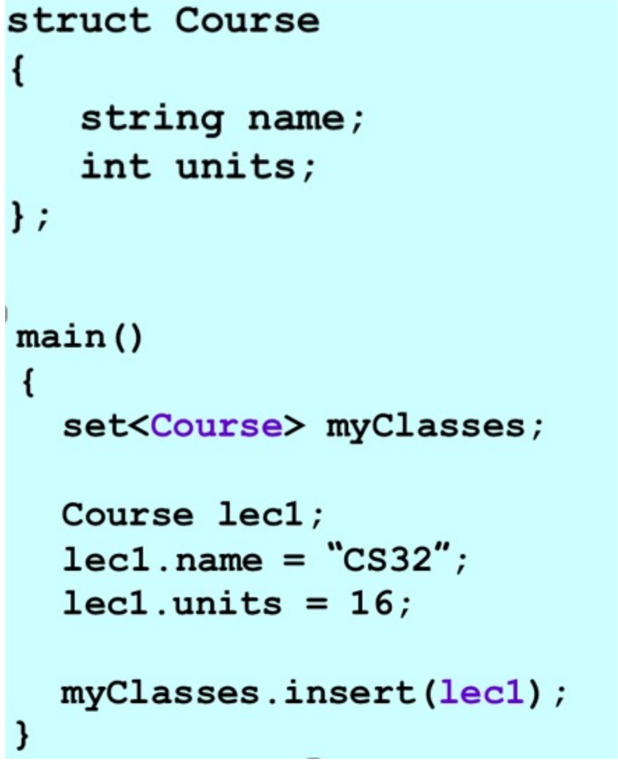
- One way association from <type1> → <type2>
- Map always maintains the keys in ordered manner
  - Operator < has to be defined for the key
- Example:

```
map<string, int> name2phone;
name2phone[‘john’] = 123456;
if(name2phone.find(‘bob’) == name2phone.end())
    cout << “Not Found!”;
```

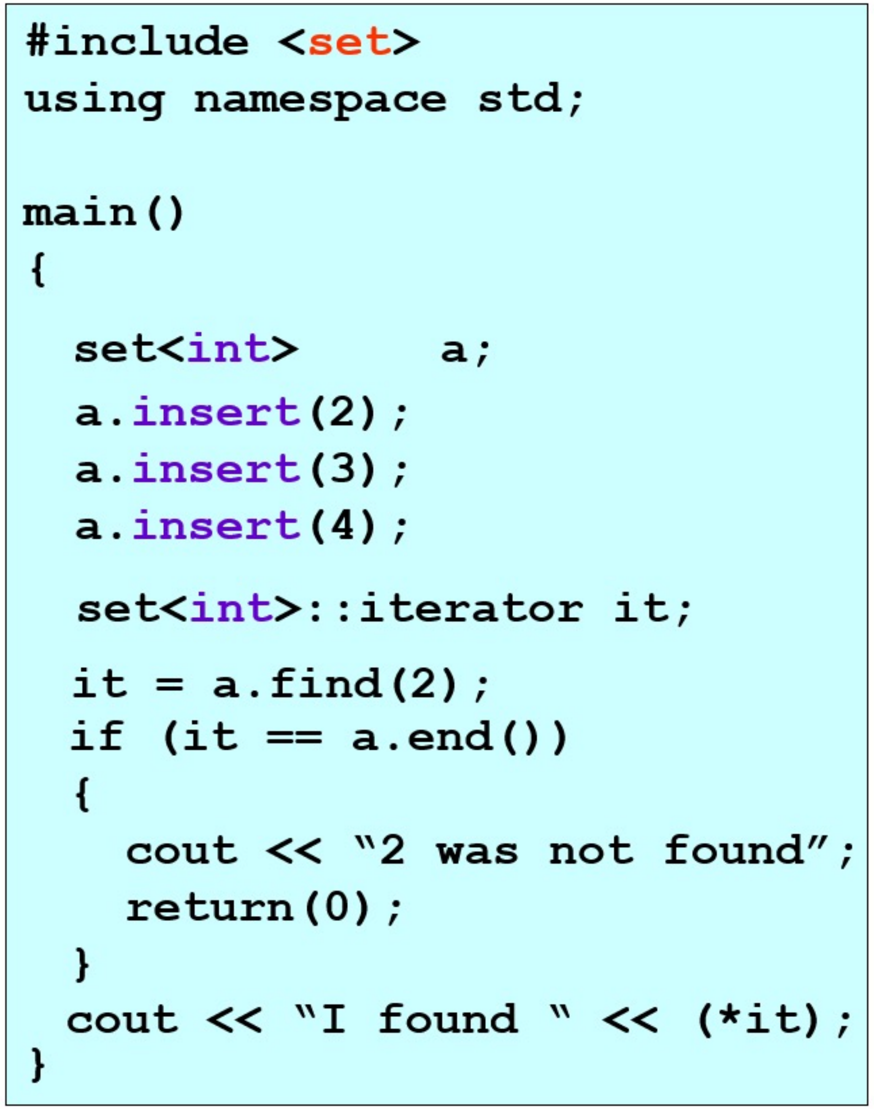
- Iterators can be used to traverse the map
  - find & end return iterators

Set

- Container that keeps of unique items
- ‘<’ operator needs to be defined



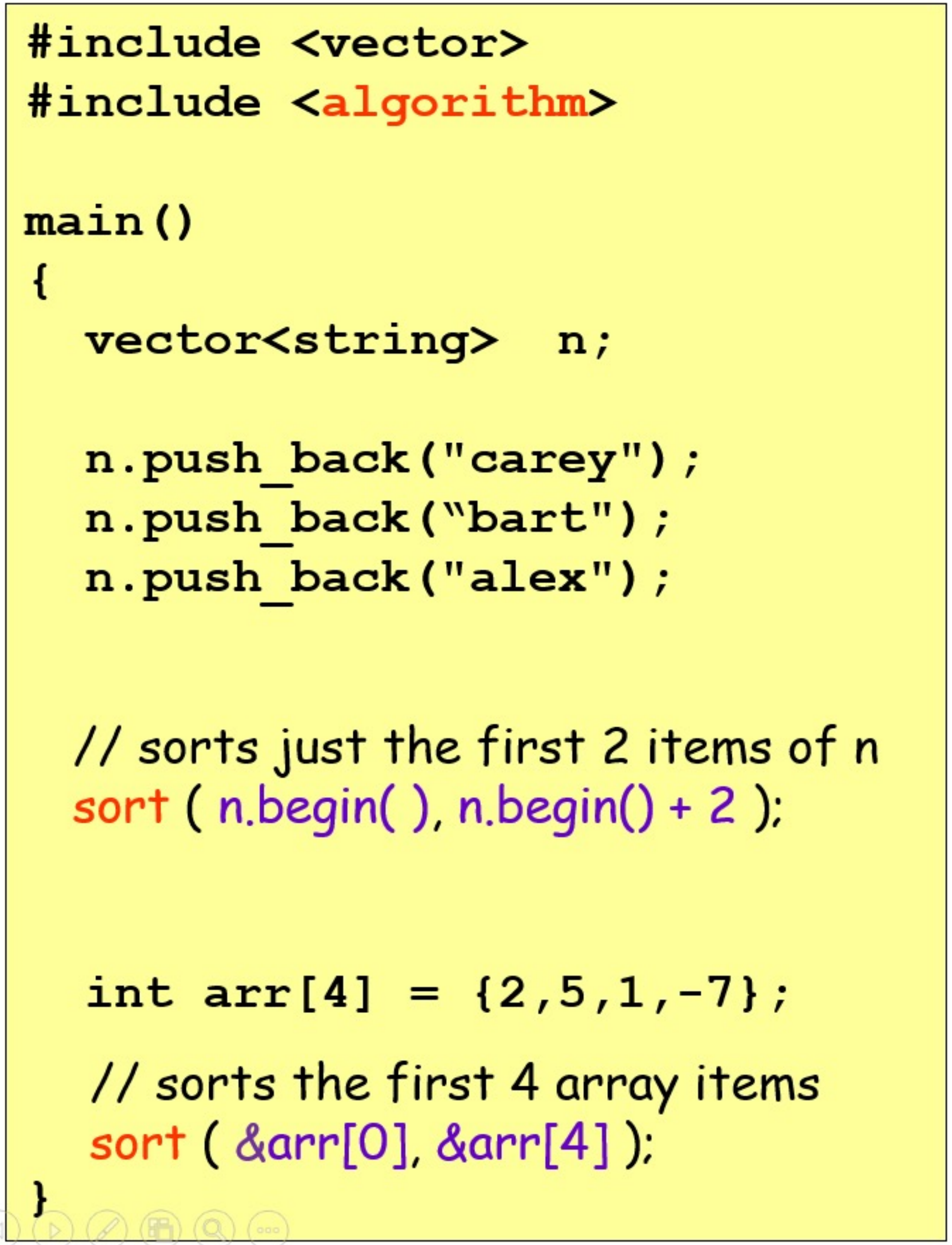
```
struct Course
{
    string name;
    int units;
};

 main()
 {
   set<Course> myClasses;

   Course lec1;
   lec1.name = “CS32”;
   lec1.units = 16;

   myClasses.insert(lec1);
}
```



```
#include <set>
using namespace std;

main()
{
  set<int>     a;
  a.insert(2);
  a.insert(3);
  a.insert(4);
  set<int>::iterator it;
  it = a.find(2);
  if (it == a.end())
  {
    cout << “2 was not found”;
    return(0);
  }
  cout << “I found “ << (*it);
}
```

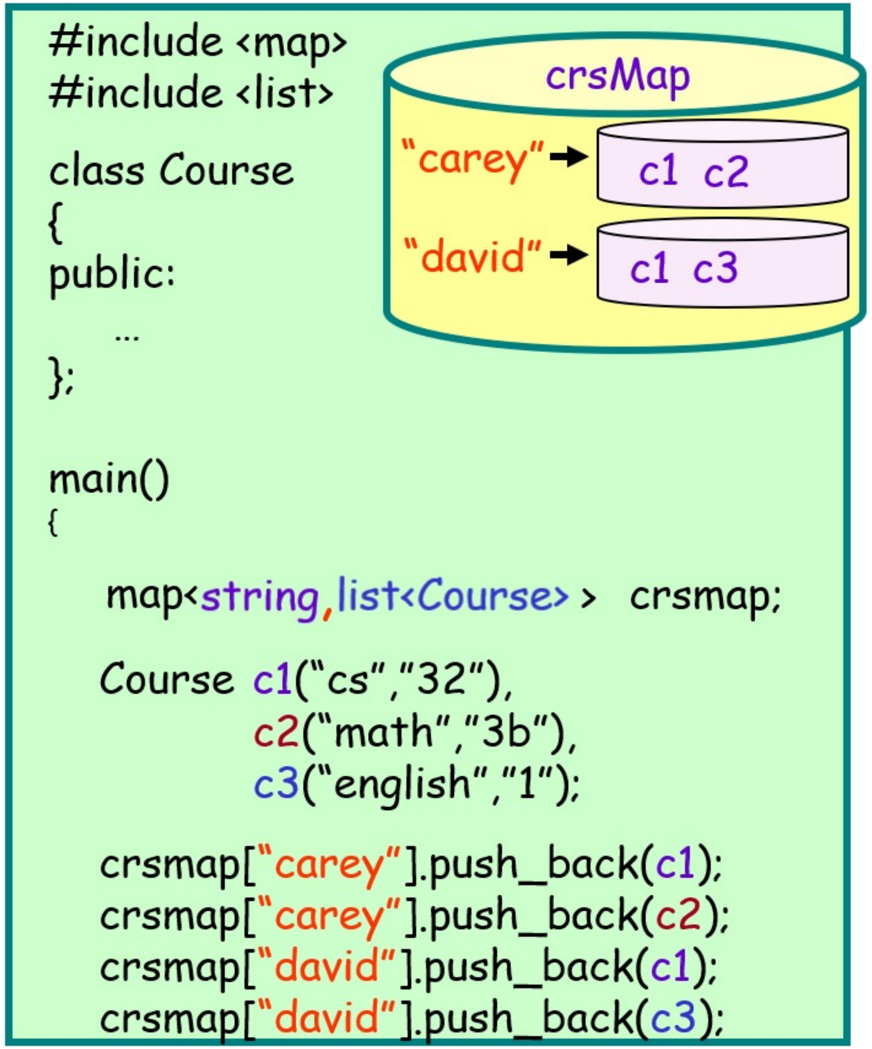
STL Algorithms

- Sorting
- Works on:
  - Arrays
  - Vectors
- *#include<algorithm>*
- Arguments:
  - 2 iterators
  - Addresses
- Can also sort objects
- Pass comparison function as third argument (must return bool)



```
#include <vector>
#include <algorithm>

main()
{
   vector<string>  n;

   n.push_back("carey");
   n.push_back(“bart");
   n.push_back("alex");


  // sorts just the first 2 items of n
  sort ( n.begin( ), n.begin() + 2 );


   int arr[4] = {2,5,1,-7};

   // sorts the first 4 array items
   sort ( &arr[0], &arr[4] );
}
```

Compound STL Data Structures



```
#include <map>
#include <list>

class Course
{
public:
     ...
};

main()
{
    map<string,list<Course> >   crsmap;

    Course c1("cs","32"),
           c2("math","3b"),
           c3("english","1");

    crsmap["carey"].push_back(c1);
    crsmap["carey"].push_back(c2);
    crsmap["david"].push_back(c1);
    crsmap["david"].push_back(c3);
```