GENERIC ABSTRACTIONS in C++

- C++ Templates
- STL (Standard Template Library)

9.4 *Programming Languages Concepts*

by John Mitchell

Overview

- Motivation
- Template review
  - Function template
  - Class template
- What is the STL?
  - Containers
  - Iterators
  - Algorithms
- Glossed-over stuff

Motivation

- Abstract data types such as stacks or queues are useful for storing many kinds of data
- It is time consuming to write different versions of stacks for different types of elements
- Most typed languages support some form of **type parameterization**
- The **C++ template** is the most familiar type-parameterization mechanism
- The **C++ STL** is a large program library of parameterized abstract data types

C++ Function Template (1)

- A simple swap function:

```
void swap(int& x, int& y){
          int tmp=x; x=y; y=tmp; }
```

- A function template with a type variable **T** in place of **int**:

```
template<class T>
       void swap(T& x, T& y){
            T tmp=x;x=y;y= tmp; }
```

nota:

```
template<typename T> =template<class T>
```

C++ Function Template (2)

- Function templates are instantiated automatically by the program linker using the types of the function arguments

```
int i,j;
...
swap(i,j); // Use swap with T replaced by int
string s,t;
...
swap(s,t); // Use swap with T replaced by String
float a;
...
swap(i,a); // ERROR
```

C++ Function Template (3)

- For each type variable, at least one function argument must depend on the type variable
  - **template<class T> T f(T &); //OK**
  - **template<class T> T f(double); //ERROR**
  - **template<class T> T f(double, T&); //OK**
  - **template<class T, class S> T f(T &, S &); //OK**
  - **template<class T, class S> T f(S &); //ERROR**

C++ Function Template (4)

- Operations on Type Parameters limit the variability of the parameters

- A generic sort function:

```
template <class T>
void sort( int count, T * A[count] ) {
  for (int i=0; i< count-1; i++)
   for (int j=i+1; j< count-1; j++)
        if (A[j] < A[i]) swap(A[i],A[j]);
}
```

- If A is an array of type **T**, then **sort(n, A)** will work only if operator **<** (possibly *overloaded*) is defined on type **T**

Esercizio

- Definiamo la funzione max tra due elementi generici.

C++ Class Template

```
template <class T> class Complex {
    private:
        T re,im;
    public:
        Complex (const T& r, const T& I)

            :re(r),im(i){}
        T getRe() {return re;}
        T getIm() {return im;}
}
```

- Type variables are fixed explicitly when the object is initialized
  - `Complex <double> x(1.0,2.0) // T = double`
  - `Complex <int> j(3,4) // T = int`
  - `Complex <char*> str("1.0","6") // T = char *`

C++ Class Template

- Type variables can be constant

```
template <class T, int dim> class Message{
private:
T mess[dim];
...
public:
Message (T *str, int n){
 int end = min(n,dim);
 for(int i=0; i<end; i++)
      mess[i]= str[i];
}
Message <char, 80> m (“Message 1”, 8);
// T = char, dim = 80
```

What is the STL?

- "Standard Template Library" by Alex Stepanov in 1976
- Basic motivation:
  - *N* data types, *M* containers, and *K* algorithms
  - Possibly *N* **M* **K* implementations
    - CountIntegerInList(IntList il, int toFind), CountIntegerInSet, CountDoubleInList, etc.
  - STL (with C++ templates): *N* +*M* +*K* implementations
    - algorithms operate over containers of types
    - `set<int> mySet;`
    - `count(mySet.begin(), mySet.end(), 4);`
    - `list<double> myList;`
    - `count(myList.begin(), myList.end(), 3.14);`

Platforms

- STL is part of Standard C++
- Porte din all the major compilers
- Stlport.org
  - Free std C++ implementation (including iostreams), some nice features/performance

STL overview

- Fundamentally, the STL defines *algorithms* that operate over a *range* in a *container*
- Our order:
  - **Containers**: a collection of typed objects
  - **Iterators** (ranges): generalization of pointer or address to some position in a container
  - **Algorithms**

Containers

- **Lists**
  - vector, list, deque
- **Adaptors**
  - queue, priority_queue, stack
- **Associative**
  - map, multimap, set, multiset
  - hash_{above}

vector<T>

- #include <vector>
- A dynamic array: random-access, grows
- Array-indexing syntax: **`operator[](dim_type n)`**

```
vector<int> v(10); v[0] = 4;
```

Defining a Vector

- Basic definition

Container's object name

```
vector<T> name;
```

Base element type

- The type can be any type or class!
- Must have: **#include <vector>**
- **Must have: using namespace std;**
- Creates an empty vector
- Example

```
vector<int> A;              // 0 ints
vector<double> B;           // 0 doubles
vector<string> C;           // 0 strings
```

Modifying a vector object

- Add a new element at the end of the vector
  - **`push_back(const T &val)`**
    - Inserts a copy of **`val`** after the last element of the vector
- Remove one element at the end of the vector
  - **`pop_back()`**
    - Removes the last element of the vector

How many elements?

- **`size_type size()`**
  - Returns the number of elements in the vector

```
cout << A.size();
```

  - Note: size_type is an “alias” name for an unsigned int

- **`bool empty()`**
  - Returns true if there are no elements in the vector; otherwise, it returns false

```
if (A.empty()) {
    // ...
```

Example vector 1

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
  vector<int> A;
  if ( A.empty() ) cout << "A has size zero. ";
  A.push_back(3); // A:  3
  A.push_back(-25); // A: 3 -25
  cout << "Size of A: " << A.size(); // size 2
  A.pop_back();  // A: 3
  cout << "Size of A: " << A.size(); // size 1
}
```

Removing All Elements

- Two member function calls to remove all elements
  - Sometimes we need to “clear out” an existing vector
- **`void resize(size_type s)`**
  - The number of elements in the vector is now **`s`**.
  - Use with zero to remove all elements
  - If you “grow” a vector, default value/constructor used for new items
- **`void clear()`**
  - Removes all elements

```
vector<int> A;
// assume we add elements to A here
A.resize(0);          // A is now empty
A.clear();            // same effect as above
```

Accessing Just One Element

- What if we want to retrieve or change one element?
  - Index value: from 0 to **`size()-1`**
  - Pass index to the **`at()`** member function
- Example:

```
vector<int> A;
// assume we add two or more elements to A
A.at(0) = A.at(1) + 1;
```

- Note: can be used on left-hand side of assignment!
  - E.g. this changes the element stored at index 0
- Example: set last element to value of 1st element

```
A.at( A.size()-1 ) = A.at(0);
```

What's Allowed on the Element?

- When you access one single element using **`at()`**, what are you allowed to do with that element?
  - **Anything** you could normally do with one variable of that type!
- Example: if **`A`** is a vector of **`int`**'s, and the element at index **`i`** exists
  - Element **`A.at(i)`** is an **`int`** just like any other **`int`** variable
    - We can print it, add to it, take its sqrt, pass it as a parameter to a function expecting an int
- Example: if **`S`** is a vector of strings, and **`S.at(i)`** exists
  - Element **`S.at(i)`** is one **`string`** object
    - We can print it, concatenate to it, call **`size`** or **`substr`** on it, pass it as a parameter to a function expecting a string

Vector Bounds Errors

- Elements only exist from index 0 to size()-1
  - Very common error to refer to **`A.at(i)`** where **`i==A.size()`**
  - If there are 10 items, the last one is at index 9
- What if you make such a *vector-bounds error*?
- The **`at()`** member function checks its parameter
  - If not in bounds, throws a run-time exception
  - Your program halts
  - (Heard of arrays? They don't do this check.)

Example 2

```
#include <vector>
#include <string>

int main() {
  int i;
  vector<string> A;
  A.push_back("I"); A.push_back("am");
  A.push_back("me");

  for (i = 0; i < A.size(); ++i) // why not <= ?
     cout << A.at(i) << " ";
  cout << endl;
```

Example 2 continued

```
    // swap 1st and last elements
    string Temp = A.at(0);
    A.at(0) = A.at( A.size()-1 ); // NOTE!!!
    A.at( A.size()-1 ) = Temp;

    A.at( A.size()-1 ) += "!"; // add ! to end

    for (i = 0; i < A.size(); ++i)
        cout << A.at(i) << " ";
    cout << endl;

    return 0;
}
```

Operating on the Whole Vector

- We can do some things on the entire vector
  - Assignment: If two vectors are defined to hold the same kinds of elements
  - Example:
  - ```
    vector<int> A, B;
    // assume we add some elements A
    B = A; // B's old contents gone, now == A
    ```

- Logical equality operators == and != work too

```
if (B == A) { // same size, same (==) elements ?
```
```

Function Examples: Input

```
void GetIntList(vector<int> &A) {
   A.resize(0);
   int Val;
   while (cin >> Val) {
       A.push_back(Val);
   }
}
```

```
vector<int> List;
cout << "Enter numbers: ";
GetIntList(List);
```

Function Example: Output

```
void PutIntList(const vector<int> &A) {
  for (int i = 0; i < A.size(); ++i) {
      cout << A.at(i) << endl;
  }
}
```

```
                  vector<int> MyList;
           // somehow values get into MyList
                  cout << "Your numbers: ";
                  PutIntList(MyList)
```

- Question: Why is formal parameter const reference?

Other Useful Functions

- Often we need to search a vector for an item:
  - **`int find (const vector<T> &vect, T target);`**
    - Loops through the elements in the vector, searching for an element equal to **`target`**
    - Returns index of **`target`** if it's found.
    - If not found, return either -1 or **`vect.size()`**
    -
- Defined functions only allow us to add/remove at vector's end
  - By using **`push_back()`** and **`pop_back()`**
  - Could we write functions that take an index value and use it to tell us where to insert or remove an element?

Other Useful Functions (cont'd)

- **void deleteAt (vector<T> &vect, int idx);**
  - Remove the element at index **idx** (if it exists)
  - How? Must use loop to "shift down" elements, then call **pop_back()** to remove unneeded element at the end

- **void insertAfter (vector<T> &vect,**
- **T newItem, int idx);**
  - Add **newItem** after element with index **idx**
  - How?
    - Must **push_back()** to get one more "space"
    - Must use loop to "shift up" elements
    - Finally do: **vect.at(idx+1) = newItem;**

vector<T>

- Time:
  - constant time insertion and removal of elements at the end
  - linear time insertion and removal of elements at the beginning or in the middle.
- The “standard” container

Forward reference: Iterators

- v.begin() and v.end() return iterators
- Like pointers: arithmetic (++, --) and dereferencing (*)

```
for (vector<int>::iterator i =
  v.begin(); i != v.end(); ++i)
  cout << *i;
```

list<T>

- Bidirectional, linear list
- Sequential access only (not L[52])
- Constructors
  - list<T>()
  - list<T>(size_t num_elements)
  - list<T>(size_t num, T init)
- Properties
  - l.empty() // true if l has 0 elements
  - l.size() // number of elements

list<T>

- Adding/deleting elements
  - l.push_back(43);
  - l.push_front(31);
  - l.insert(iterator,4) // insert 4 before the position “iterator”
  - etc..
- Accessing elements
  - l.front() // T &
  - l.back() // T &
  - l.begin() // list<T>::iterator
  - l.end() // list<T>::iterator

list<T>

- Removing elements
  - l.pop_back() // returns nothing
  - l.pop_front() // returns nothing
  - l.erase(iterator i)
  - l.erase(iter start, iter end) // delete a *range*
- Time
  - Amortized constant time insertion and removal of elements at the beginning or the end, or in the middle [because you pass an iterator]

list<T>

- Other operations
  - l.sort(), l.sort(CompFn) // sorts in place
  - l.splice(iter b, list<T>& grab_from)

list<T>

- Example:

```
list<char> l;
for (int i = 0; i < 4; ++i)
{
  l.push_front(i + 'A');
  l.push_back(i + 'A');
}
for (list<char>::iterator i = l.begin();
  i != l.end(); ++i)
  cout << *i; // DCBAABCD
```

Other data structures

- Hashtables / Map
- Queue
- Stack
- Set
- …
- algorithms …

[hash_]map, [hash_]multimap

- A map is an “associative container”
- Given one value, will find another
  - map<string, int> is a map from strings to int’s
  - maps are 1:1, multimap are 1:n
- map, multimap are **logarithmic** when inserting/deleting
  - Needs to maintain sortedness
- hash_map, hash_multimap are amortized **constant time**
  - Not sorted (“hashed”)

Map functions

- m.insert(make_pair(key, value)); // inserts
- m.count(key); // times occurs (0, 1)
- m.erase(key); // removes it
- m[key] = value; // inserts it into the table
- m[key] //retrieves or creates a “default” for it
- i=m.begin(), i=m.end() // iterators
- i->first, i->second // per accedere a chiave e valore della coppia puntata da i

Hash_{…}

- There are hash_map, hash_multimap, hash_set, hash_multiset
- Basically, these are constant time insert/delete instead of log time
  - They don’t maintain sortedness
  - Me: reduced running time from 10 min to 5 min

Hash performance

- Fill with 100,000 random elements
- Lookup 200,000 random elements
  - Same random seed
- map: fill 0.59967s
- map: lookups 1.57483s
- hash_map: fill 0.615407s
- hash_map: lookups 0.872557s
- So, if you don’t need order, go with hash_map

Summary

- map: 1:1, sorted, m[k] = v
- multimap: 1:n, sorted, mm.insert(make_pair(k,v))
- set: unique elements, sorted
- multiset: multiple keys allowed, sorted
- hash_: faster but **not sorted**

Iterators

- Touched on earlier
- An iterator is like a pointer
- You can increment to it to go to the “next” element
- You can [sometimes] subtract or add N
- You can dereference it
- Different kinds of iterators
- Most useful when combined with algorithms

Iterators

- c.begin() = start
- c.end() = 1 past the last element
  - Never dereference end! (*c.end() is bad!)
- Why? Makes loops simpler.
- Prefer ++i because i++ makes a temporary object and returns it, incrementing later.

Different kinds

- Technically:
  - random access (i += 3; --i; ++i)
  - bidirectional (++i, --i), store/retrieve
  - forward (++i), store/retrieve
  - input (++i) retrieve
  - output (++o) store
- But, writing code directly using iterators hurts a lot

Practical iterators

- iterator
  - “Standard”, goes from beginning to end
  - c.begin(), c.end()
- const_iterator
  - Like iterator, but changes can’t be made (prefer!)
  - c.begin() and c.end() are overloaded so you can use them to assign their result to a const_iterator
- reverse_iterator
  - Goes from the end to the beginning with same semantics as iterator
  - Generally, c.rbegin() and c.rend()
  - list, vector, deque, map, multimap, set, multiset, hash_, string

Iterator example

```
vector<int> v;
for (int k = 0; k < 7; ++k) v.push_back(k);
display(v); // 0 1 2 3 4 5 6

for(vector<int>::iterator i = v.begin(); i != v.end();
   ++i)
   *i = *i + 3; // add three to content
display(v); // 3 4 5 6 7 8 9

for(vector<int>::const_iterator ci = v.begin();
 ci != v.end(); ++ci)
   cout << *ci << ' ';// *ci = *ci - 3; won't compile
cout << endl;//    3 4 5 6 7 8 9

for (vector<int>::reverse_iterator ri = v.rbegin();
  ri != v.rend(); ++ri)
{  *ri = *ri - 3;
   cout << *ri << ' ';}
cout << endl; //6 5 4 3 2 1 0
```

Sort Functions

- Just a touch!

```
vector<int> v;
// fill v with 3 7 5 4 2 6
sort (v.begin(), v.end() );
```

Polymorphic STL containers and iterators

- **STL and inheritance do not mix well together**
- A STL container expects to contain its objects directly, so:

```
ellipse e(rect1);
rectangle r(rect2);
list < shape > shapeList;
shapeList.push_back(e);
shapeList.push_back(r);
```

- this code will compile but will do a slicing to shapes.

(2)

- It becomes apparent that one level of indirection is needed to solve the problem. An obvious solution is to change the list of shapes to a list of pointers to shapes:
- list < shape* > shapeList;
- and list would be populated:
- shapeList.push_back(&e);
- shapeList.push_back(&r);
- Other problems arise...

Esempio

- Lista di studenti con due classi derivate LS e IL …
- Implementa un metodo (virtual?) calcolo media
- Fai un vector di Studenti
- Inserisci tre studenti
- Chiama per tutti in metodo calcolo media

Conclusion

- The STL has everything
- Let the compiler do the work for you
- Saves time and lines of code
- **Run-time efficiency** of the code that is generated
- Next steps:
  - Buy a good book on STL
    - Schildt's STL Programming from the Ground Up
  - Use it on your homeworks/personal projects
  - Learn about function objects
    - Didn't have time to cover them; another talk??

Resources

- Books
  - Schildt – “STL Programming from the Ground Up” ***
  - Schildt – “C/C++ Programmers Reference”
- URLs
  - http://www.stlport.org/resources/StepanovUSA.html
  - http://www.usenix.org/publications/library/proceedings/coots97/full_papers/sundaresan/sundaresan_html/node2.html
  - MSDN
  - Google: sgi stl <container or algorithm>