

# **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

# 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);`

---

# Where did it come from?

- Alex Stepanov
    - “In 1976, still back in the USSR, I got a very serious case of food poisoning from eating raw fish.”
    - “While in the hospital, in the state of delirium, I suddenly realized that the ability to add numbers in parallel depends on the fact that addition is associative.” (Huh?)
    - **“Putting it simply, STL is the result of a bacterial infection.”** (That I can understand.)
-

---

# Platforms

- STL is part of Standard C++
  - In Visual C++/Studio 6.0
    - Missing some stuff: `hash_map`
  - In Visual Studio.NET / VC++ 7.0
    - Still some issues
  - G++ 3.0: dunno
  - [Stlport.org](http://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

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

Container's object 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
-

---

# Exercises -

1. **STL1:** declare a vector of integer values, stores five arbitrary values in the vector and then print the single vector elements to cout.
  2. **STL2 :** declare a vector of string values, asks to the user to insert a sentence of one or more words, store each word in the vector and then print the sentence in reverse order
-



# vector<T> example

```
vector<char> v;  
for (int i = 0; i < 10; ++i)  
    v.push_back('A' + i);  
cout << v[0] << v.back() << endl; // AJ  
v.pop_back(); // doesn't return anything  
cout << v.size() << v.back() << endl;  
    // 9I  
for (size_t i = 0; i < v.size(); ++i)  
    cout << v[i]; // ABCDEFGHI (no J)  
cout << endl;
```

---

# 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;
```



# Exercise

- **STL3**: write STL2 with the iterator

```
vector<int>::iterator i = v.end();  
vector<int>::iterator first =  
    v.begin();  
while(i != first) {  
    --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**
-

# Exercise STL5 (hash\_map)

- Costruisci l'anagrafica dei voti (interi) di una classe (con nomi unici) come hash\_map come associazione  
nomi->voti
- **inserimento**: chiedi nome e voto e inserisci il dato
- **elenco**: stampa elenco nomi, voti
- **interrogazione**: chiedi il nome e stampa il voto associato

---

# 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() );
```

---

# Exercise STL 4 (List)

- Write a STL program that takes an arbitrary sequence of binary digits (integer values 0 and 1) from cin and stores them into a container. When receiving a value different from 0 or 1 from cin stop reading.
- Now, you should have a container storing a sequence of 0's and 1's. After finishing the read-process, apply a "bit-stuffing" algorithm to the container. Bit-stuffing is used to transmit data from a sender to a receiver. To avoid bit sequences in the data, which would erroneously be interpreted as the stop flag (here: 01111110), it is necessary to ensure that six consecutive 1's in the data are splitted by inserting a 0 after each consecutive five 1's.

---

# Hint!

- Get an iterator to the first list element. As long as this iterator is different from the `end()` iterator increment the iterator and dereference it to get the appropriate binary value.
  - Note that an element is always inserted before a specified iterator-position and that this insertion doesn't affect all the other iterators defined when using a list.
-

---

# 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/fu>
  - MSDN
  - Google: `sgi stl <container or algorithm>`
-

---

# Example 1

- Read in an arbitrary number of integers ( $n \geq 1$ ) from “numbers.txt” and display:
    - Minimum, maximum
    - Median
    - Average
    - Geometric mean  $((y_1 * y_2 * \dots * y_n)^{1/n})$
  - How many lines would it take you?
    - arbitrary storage (for median), sorting, loops...
-

# Example 1: STL solution

```
#include <vector>           // include STL vector implementation
#include <iostream>
using namespace std;

vector<int> v;

copy(istream_iterator<int>(ifstream("numbers.txt")),
     istream_iterator<int>(), back_inserter(v));

sort(v.begin(), v.end());
cout << "min/max: " << v.front() << " " << v.back() << endl;
cout << "median : " << *(v.begin() + (v.size()/2)) << endl;
cout << "average: " << accumulate(v.begin(), v.end(), 0.0)
     / v.size() << endl;
cout << "geomean: " <<
     pow(accumulate(v.begin(), v.end(), 1.0, multiplies<double>()),
         1.0/v.size()) << endl;
```

---

# Example 2

- Write a program that outputs the words and the number of times it occurs in a file (sorted by word)
    - File input, hashtable, hash function...
-

# Example 2: STL solution

```
vector<string> v;
```

```
map<string, int> m;
```

```
copy(istream_iterator<string>(ifstream("words.txt")),  
     istream_iterator<string>(), back_inserter(v));
```

```
for (vector<string>::iterator vi = v.begin();  
     vi != v.end(); ++vi)  
    ++m[*vi];
```

```
for (map<string, int>::iterator mi = m.begin();  
     mi != m.end(); ++mi)  
    cout << mi->first << ": " << mi->second << endl;
```