Today's Goals

- The Standard Template Library

History

- Alexander Stepanov in the late 70's
- Many algorithms do not depend on particular implementation of a data structure
- Only few fundamental semantic properties of a data structure are important
  - to iterate
  - to compare
- Algorithms can be defined as general as possible without losing efficiency

Development History

- In 1985 Stepanov developed a generic Ada library
- In 1988 Stepanov moved to HP labs
- In 1992 he was appointed manager of algorithms project, part of with was the huge library – the Standard Template Library for C++, together with Meng Lee

Standard Template Library

- The STL is a generic library
- The contents are heavily parameterized
- The parameterization is based on templates
- Main components:
  - Containers
  - Iterators
  - Algorithms
  - Functors

Templates

- Patterns from which classes can be created
- Leaves part of the class definition unspecified
- This omission makes the class more general and easier to reuse

```
class Stack {
public:
  Stack(int s);
  bool is_empty();
  void push(int);
  int pop();
...
}
```

```
template <class T>
class Stack {
public:
  Stack(int s);
  bool is_empty();
  void push(T);
  T pop();
...
}
```

Templates

- We don't just want containers of double
- We want containers with element types we specify
  - **container<double>**
  - **container<int>**
  - **container<Month>**
  - **container<Record*>** *// container of pointers*
  - **container<container<Record>>** *// container of containers*
  - **container<char>**
- We must make the element type a parameter to **container**
- **container** must be able to take both built-in types and user-defined types as element types
- This is not some magic reserved for the compiler; we can define our own parameterized types, called "templates"

Templates

- The basis for generic programming in C++
  - Sometimes called "parametric polymorphism"
    - Parameterization of types (and functions) by types (and integers)
  - Unsurpassed flexibility and performance
    - Used where performance is essential (*e.g.,* hard real time and numerics)
    - Used where flexibility is essential (*e.g.,* the C++ standard library)

- Template definitions

```
template<class T, int N> class Buffer { /* ... */ };
template<class T, int N> void fill(Buffer<T,N>& b) { /* ... */ }
```

- Template specializations (instantiations)

```
// for a class template, you specify the template arguments:
Buffer<char,1024> buf;          // for buf, T is char and N is 1024

// for a function template, the compiler deduces the template arguments:
fill(buf);      // for fill(), T is char and N is 1024; that's what buf has
```

Parameterize with element type

```
// an almost real container of Ts:
template<class T> class container {
    // ...
};
container<double> vd;                   // T is double
container<int> vi;                      // T is int
container<container<int>> vvi;          // T is container<int>
                                        //      in which T is int
container<char> vc;                     // T is char
container<double*> vpd;                 // T is double*
container<container<double>*> vvpd;     // T is container<double>*
                                        //      in which T is double
```

Template Instantiation

- Class templates are later instantiated by filling in the template arguments
- It does not have to be a class, any C++ type will do

```
template <class T>
void Stack<T>::push(T x){
...
}
```

```
Stack<int> int_stack(10);
int_stack.push(5);
Stack<double> double_stack(10);
double_stack.push(5.2);
Stack<Shape> shape_stack(10);
shape_stack.push(circle);
```

Function Objects

- Also known as functors
- Allows an object to be invoked as if it were an ordinary function, usually with the same syntax
- The STL includes classes that overload the function-call operator (`operator()`).
- These are very important in using the STL

Example

```
class Less {
public:
  Less (int v) : val (v) {}
  bool operator () (int v) { return v < val; }
private:
  int val;
};

Less less_than_5 (5);
cout << "2 is less than 5: " <<
less_than_5(2) ? "true" : "false");
```

Containers

- Sequence containers
  - **`vector`** – array with unlimited size
  - **`list`** – doubly linked list
  - **`deque`** – vector and linked list

```
vector<int> v;
v[0] = 7;
v[1] = v[0] + 3;
v.insert(v.begin(), 3);
```

```
list<int> l;
l.push_back(0);
l.push_front(1);
l.insert(++l.begin(),2);
```

```
deque<int> q;
q[0]=3;q.push_front(1);q.insert(q.begin()+1,2);
q[2]=0;
```

Iterators

- What exactly do **`v.begin()`**, **`l.begin()`**, **`q.begin()`** return?
- Iterators are a generalization of pointers
- Iterators are pointers to container items
- A mechanism to step through a container, any container
- Iterators and templates together make it possible to decouple algorithms from particular containers

Linear Search

```
template <class InputIterator, class T>
InputIterator find(InputIterator first,
InputIterator last, const T& value) {
  while (first != last && *first != value)
    ++ first;
  return first;
}
```

```
list<int> l;
list.push_back(7); // insert other numbers ...
list<int>::iterator result;
result = find(l.begin(), l.end(), 7);
```

- **`InputIterator`** is a template parameter, not a actual type

Associative Containers

- Sorted associative containers
  - **`set`** – stores unique objects of type **`Key`**
  - **`multiset`** – allows identical elements
  - **`map`** – associates objects of type **`Key`** to objects of type **`Data`**
  - **`multimap`** – non unique version of **`map`**

`set` Example

```
struct ltstr {
  bool operator()(const char* s1,const char* s2) const
  { return strcmp(s1, s2) < 0; }
}
int main() {
  const int N = 3;
  const char* a[N] = {"isomer", "ephemeral", "prosaic"};
  const char* b[N] = {"flat", "this", "artichoke"};
  set<const char*, ltstr> A(a, a + N);
  set<const char*, ltstr> B(b, b + N);
  set<const char*, ltstr> C;
  set_union(A.begin(), A.end(), B.begin(), B.end(),
  inserter(C, C.begin()), ltstr());
  set_intersection(...);
  set_difference(...);
```

map Example

```
map<const char*, int, ltstr> months;
months["january"] = 31;
months["february"] = 28;
...
months["november"] = 30;
months["december"] = 31;
map<const char*, int, ltstr>::iterator cur;
cur = months.find("june");
cout <<(*cur).first<<" has "<<(*cur).second<<" days ";
cout << (*(cur+1)).first << " has "
<< (*(cur+1)).second << " days" << endl;
```

Hashed Containers

- Hashed Associative containers
  - **`hash_set`**
  - **`hash_multiset`**
  - **`hash_map`**
  - **`hash_multimap`**
- Equivalents of the associative containers before, stored in hash tables

`hash_set` Example

```
struct eqstr {
  bool operator()(const char* s1, const char* s2) const
  { return strcmp(s1, s2) == 0; }
};

void lookup(const hash_set<const char*, hash<const
char*>, eqstr>& Set, const char* word) {

hash_set<const char*, hash<const char*>,
eqstr>::const_iterator it = Set.find(word);

cout << word << ": " << (it != Set.end() ? "present" :
"not present") << endl;
}
```

`hash_set` Example

```
int main() {
  hast_set<const char*,hash<const char*>,eqstr> fruits;
  fruits.insert("kiwi");
  fruits.insert("plum");
  fruits.insert("mango");
  fruits.insert("banana");
  fruits.insert("apple");

  lookup(fruits, "mango");
  lookup(fruits, "apricot");
```

Algorithms

- STL includes a large collection of built-in algorithms that manipulate containers
  - `reverse(v.begin(), v.end());`
  - `find(l.begin(), l.end(), 7);`

Non-mutating

- `for_each`
- `find, find_if, adjacent_find, find_first, find_end`
- `count, count_if`
- `equal`
- `mismatch`
- `search, search_n`

Mutating

- **`copy, copy_n, copy_backward`**
- **`swap, iter_swap, swap_ranges`**
- **`replace, replace_if, replace_copy, replace_copy_if`**
- **`fill, fill_n`**
- **`generate, generate_n`**
- **`remove, remove_if, remove_copy, remove_copy_if`**

Mutating

- **`unique, unique_copy`**
- **`reverse, reverse_copy`**
- **`rotate, rotate_copy`**
- **`random_shuffle, random_sample, random_sample_n`**
- **`partition, stable_partition`**

Sorting

- Sort
  - `sort, stable_sort, partial_sort, partial_sort_copy, is_sorted`
  - `nth_element`
- Binary search
  - `lower_bound, upperbound, equal_range, binary_search`
- Merge
  - `merge, inplace_merge`
- Permutations
  - `next_permutation, prev_permutation`

Sorting

- Set operations on sorted ranges
  - **`includes, set_union, set_intersection, set_difference, set_symmetric_difference`**
- Heap
  - **`push_heap, pop_heap, make_heap, sort_heap, is_heap`**
- Minmax
  - **`min, max, min_element, max_element`**
- Lexicographical
  - **`lexicographical_compare, lexicographical_compare_3way`**

Generalized numeric algorithms

- `iota`
- `accumulate`
- `inner_product`
- `partial_sum`
- `adjacent_difference`
- `power`

Other Algorithms

- Function Objects
- Utilities
- Memory Allocation

Summary

- Most STL container types require **`#include`**
- C++ string is part of STL
- STL is incredibly useful
- Using STL requires a good understanding of abstract data types and template parameterization