HEWLETT PACKARD

The Standard Template Library

Alexander Stepanov

Meng Lee

Hewlett Packard Laboratories

1501 Page Mill Road, Palo Alto, CA 94304

stepanov@hpl.hp.com & lee@hpl.hp.com

March 7, 1994

The design principles:

1. Comprehensive

- *take all the best from APL, Lisp, Dylan, C library, USL Standard Components...*
- *provide structure and fill the gaps*

2. Extensible

- *orthogonality of the component space*
- *semantically based interoperability guarantees*

3. Efficient

- *no penalty for generality*
- *complexity guarantees at the interface level*

4. Natural

- *C/C++ machine model and programming paradigm*
- *support for built-in data types*

Component Classification:

- container — manages a set of memory locations
- iterator — provides a traversal protocol through a container
- algorithm — encapsulates a computational process
- applicative object — encapsulates a state (possibly empty) together with an algorithm

Merge

```
int a[1000];
int b[2000];
int c[3000];
...
mergeCopy(a, a + 1000, b, b + 2000, c);


Vector<int> x;
List<int> y;
int z[...]
...
mergeCopy(x.begin(), x.end(), y.begin(), y.end(), z);
```

MergeCopy(1)

```
template <class Iterator1, class Iterator2, class ResultIterator>
ResultIterator mergeCopy(Iterator1 first, Iterator1 last,
                 Iterator2 otherFirst, Iterator2 otherLast,
                 ResultIterator result) {
    while (first != last && otherFirst != otherLast)
        if (*otherFirst < *first)
            *result++ = *otherFirst++;
        else
            *result++ = *first++;
    return copy(otherFirst, otherLast, copy(first, last, result));
}
```

MergeCopy(2)

```
template <class Iterator1, class Iterator2, class ResultIterator,
          class Compare>
ResultIterator mergeCopy(Iterator1 first, Iterator1 last,
                 Iterator2 otherFirst, Iterator2 otherLast,
                 ResultIterator result, Compare comp) {
    while (first != last && otherFirst != otherLast)
        if (comp(*otherFirst, *first))
            *result++ = *otherFirst++;
        else
            *result++ = *first++;
    return copy(otherFirst, otherLast, copy(first, last, result));
}
```

Intmerge

```
#include <stl.h>

main(int argc, char** argv) {
    if (argc != 3) throw("usage: intmerge file1 file2\n");
    mergeCopy(InputIterator<int>(ifstream(argv[1])), InputIterator<int>(0),
              InputIterator<int>(ifstream(argv[2])), InputIterator<int>(0),
              OutputIterator<int>("\n"));
}
```

Deterministic sort: free reference implementation of sort

```
template <class BidirectionalIterator>
void deterministicSort(BidirectionalIterator first,
                       BidirectionalIterator last) {
    while (nextPermutation(first, last));
}


template <class Iterator, class Compare>
void deterministicSort(BidirectionalIterator first,
                       BidirectionalIterator last, Compare comp) {
   while (nextPermutation(first, last, comp));
}
```

Partial sort

```
#include <stl.h>

// prints n smallest integers from stdin
main(int argc, char** argv) {
    if (argc != 2) throw("usage: partialsort number\n");
    Vector<int> v(size_t(atoi(argv[1])), 0);
    copy(v.begin(),
        partialSortCopy(InputIterator<int>(), InputIterator<int>(0),
                         v.begin(), v.end()),
        OutputIterator<int>("\n"));
}
```

Sort

```
#include <stl.h>
#include "string.H"

// sorts a file lexicographically
main(int argc, char**) {
    if (argc != 1) throw("usage: sort\n");
    Vector<String> v;
    copy(InputIterator<String>(), InputIterator<String>(0),
         VectorInsertIterator<String>(v, v.end()));
    sort(v.begin(), v.end());
    copy(v.begin(), v.end(), OutputIterator<String>());
}
```

Sequence containers:

Vector:

random access

constant time (amortized) *insert* and *erase* at the end

List:

sequential access

constant time *insert* and *erase* anywhere

Deque:

random access (but slower than Vector)

constant time *insert* and *erase* at the beginning and the end

All three share the same interface

Taxonomy of iterators

Primary

- Forward iterator
- Bidirectional iterator
- Random access iterator

Additional

- Iterator, result iterator - weaker versions of forward iterators
- Insert iterator - special kind of result iterator
- Bidirectional reverse iterator, random access reverse iterator - reverse iterators

NOTE: ALL THESE ARE NOT CLASSES BUT CLASS REQUIREMENTS

Iterator template classes:

Random access iterators:

(Pointer), ReversePointer, DequeIterator, DequeReverseIterator

Bidirectional iterators:

ListIterator, ListReverseIterator

Iterators:

InputIterator

Result iterators:

OutputIterator, VectorInsertIterator, ListInsertIterator, DequeInsertIterator,

InsertPointer

Applicative objects:

```
cout << Greater<int>()(27, 5);
prints: true

cout << GreaterX<int>(5)(27); // bind the second argument
prints: true

cout << XGreater<int>(27)(5); // bind the first argument
prints: true
```

Example:

```
int a[10000];
...
sort(a, a + 10000, Greater<int>()); // sort in descending order
```

Applicative template classes

Identity

Binary arithmetic operations:

Plus, Minus, Times, Divides, Modulus (with X<op> and <op>X)

Unary arithmetic operations: Negate

Binary relational operations:

Equal, NotEqual, Greater, Less, GreaterEqual, LessEqual (with X<op> and <op>X)

Unary relational operations: Not

Increment & decrement:

PrefixPlusplus, PostfixPlusplus, PrefixMinusminus, PostfixMinusminus

Assignment

Trivial predicates: True, False

Algorithmic template functions:

Swap

Functions on ordered elements: min, max

General Iterations: forEach, accumulate, innerProduct

Searching: find, adjacentFind, mismatch, equal, search, count

Order selectors: maxElement, minElement

Lexicographical comparisons: lexicographicalCompare

Searching in sorted structures: lesserRange, greaterRange, equalRange, isMember

Transformers: copy, copyBackward, swapRanges, transform, transformCopy, replace, replaceCopy, partialSum, partialSumCopy, adjacentDifference, adjacentDifferenceCopy, fill, iota

Removers: remove, removeCopy, unique, uniqueCopy

Merging of sorted structures: merge, mergeCopy

Set operations on sorted structures: includes, unionCopy, intersectionCopy, differenceCopy, symmetricDifferenceCopy

Permutations: reverse, reverseCopy, rotate, rotateCopy, randomShuffle

Permutation generators: nextPermutation, prevPermutation

Partitions: partition, stablePartition

Sorting: sort, stableSort, partialSort, partialSortCopy, select

Heap operations: pushHeap, popHeap, makeHeap, sortHeap

List mutative operations: listRemove, listUnique, listMerge, listReverse, listSort

Memory management

```
template <class T>
T* allocate(size_t n, T*);

template <class T>
void deallocate(T* buffer);

template <class T>
Pair<T*, size_t> getTemporaryBuffer(size_t n, T*);

template <class T>
void construct(T* p, const T& value);

template <class T>
void destroy(T* pointer);

template <class T>
void destroy(T* first, T* last);
```

Conclusions:

1. HP position:

- make the library widely available
- free reference implementation and validation suite are under consideration

2. The status of the library:

- fully implemented under HP C++ compiler
- tested, the full validation suite is under construction
- porting to other compilers is under way