HEWLETT PACKARD

Generic C++ Components

Mehdi Jazayeri, Meng Lee, and Alex Stepanov
Hewlett-Packard Laboratories

October 4, 1993

Outline of talk

1. An outrageous claim
2. Code walk
3. Some theory
4. Conclusions

We have libraries of software components.

The components are:

- useful
- efficient
- flexible
- correct
- tested
- measured

The libraries are:

- comprehensive
- structured
- documented

```
#include <file_handling.H>
#include <vector.H>
#include <pair.H>
#include <lexical.H>
#include <random_access_sort.H>
#include <iter_ostream.H>

typedef Pair<char*, size_t> Line;

main(int, char** argv)
{
        Extent input(argv[1]);
        Vector<Line> vec;
        makeLineIndex(input.begin(), input.end(), vec, '\n');
        quickSort(vec.begin(), vec.end(), LineCompare<Line>());
        streamLineIndex(vec.begin(), vec.end(), IterOstream<char>(cout));
}
```

```
template <class Iterator, class Container, class Recognizer>
void tokenize(Iterator first, Iterator last, Container& v, Recognizer machine)
{
        for (; first != last; first++)
            tokenInfoUpdate(first, v, machine(*first));
}

template <class Iterator, class Container, class TokenInfo>
inline void tokenInfoUpdate(Iterator position, Container& v, TokenInfo info)
{
        if (size_t(info) == 1)
            v.insertAtEnd(makePair(position, info));
        else
            (*(v.end() - 1)).second = info;
}
```

```
template <class T>
class EndScan
{
protected:
        T element;
        size_t n;
public:
        EndScan(T x) : element(x), n(0) { }
        size_t operator()(T x)
        {
            size_t tmp = ++n;
            if (x == element) n = 0;
            return tmp;
        }
};
```

```
template <class Iterator, class Container, class T>
void makeLineIndex(Iterator first, Iterator last, Container& v, T delimeter)
{
        tokenize(first, last, v, EndScan<T>(delimeter));
}

template <class Iterator1, class Iterator2>
void streamLineIndex(Iterator1 first, Iterator1 last, Iterator2 result)
{
        for (; first != last; first++)
            result = move((*first).first, size_t((*first).second), result);
}
```

```
template <class Pair>
class LineCompare
{
public:
        LineCompare() { }
        int operator()(Pair i, Pair j)
        {
            return lexicographicalDifference(i.first, i.first + size_t(i.second),
                                             j.first, j.first + size_t(j.second));
        }
};
```

```
template <class Iterator1, class Iterator2>
inline Iterator2 move(Iterator1 first, Iterator1 last, Iterator2 result)
{
        while (first != last) *result++ = *first++;
        return result;
}

template <class Iterator1, class Iterator2>
inline Iterator2 move(Iterator1 first, size_t n, Iterator2 result)
{
        while (n--) *result++ = *first++;
        return result;
}
```

```
template <class Iterator1, class Iterator2>
int lexicographicalDifference(Iterator1 first1, Iterator1 last1,
                              Iterator2 first2, Iterator2 last2)
{
    while (first1 != last1 && first2 != last2) {
        int tmp = *first1++ - *first2++;
        if (tmp != 0) return tmp;
    }

    if (first1 != last1)
        return 1;
    else if (first2 != last2)
        return -1;
    else
        return 0;
}
```

```
template <class Iterator1, class Iterator2, class Compare>
int lexicographicalDifference(Iterator1 first1, Iterator1 last1,
                              Iterator2 first2, Iterator2 last2,
                              Compare comp)
{
    while (first1 != last1 && first2 != last2) {
        int tmp = comp(*first1++, *first2++);
        if (tmp != 0) return tmp;
}
    if (first1 != last1)
        return 1;
    else if (first2 != last2)
        return -1;
    else
        return 0;
}
```

Component programming

Generic algorithms X Generic data structures X Data types

Requires

- syntactic uniformity: C++ operator overloading, template functions, ...
- *semantic* uniformity: object algebra: set of axioms and theorems for a related family of classes

NICE CLASSES

(joint work with Andrew Koenig - Bell Labs)

class T is called "nice" iff it supports:

- T(T&)
- T& operator=(T&)
- int operator==(T&)
- int operator!=(T&)

such that:

1. T a(b); assert(a == b);
2. a = b; assert(a == b);
3. a == a
4. a == b iff b == a
5. (a == b) && (b == c) implies (a == c)
6. a != b iff !(a == b)

NICE CLASSES (2)

A member function T::s(...) is called *equality preserving* iff

a.s(...) == b.s(...)

A class is called *Extra-nice* iff

all of its member functions are equality preserving

EQUALITY FOR CONTAINERS

Both size and dereferencing of the iterators are equality-preserving.

```
Container<T> a(b);
assert(a.size() == b.size());
```

Moreover, for any valid Iterator type for Container

```
Iterator<T> x = a.begin();
Iterator<T> y = b.begin();
assert(*x.advance(n) == *y.advance(n));
```

(advance(...) is any iterator-moving function)

CLASSIFICATION OF ITERATORS

Iterators are *extra-nice* classes with operator*() defined.

- *readable* iterator: *i returns a rvalue of type T
- *writable* iterator: *i returns a lvalue which takes T
- ***regular*** iterator: **both** readable and writable
- ***trivial*** iterator: no moves
- ***sequential*** iterator: ++
- ***bi-directional*** iterator: --
- ***full sequential*** iterator: +=(int) — constant time!

CONCLUSIONS

- We have over 500 components now
- Will have them tested and fully documented by January 94
- You should use them
- HP should sell them