STL and Its Design Principles

Alexander Stepanov

Outline of the Talk

- Genesis of STL
- Fundamental principles
- Language requirements
- Industrialized software engineering
- Assessment

Genesis of STL

- Original intuition: associating algorithms with mathematical theories – 1976
- Specification language *Tecton* (with Dave Musser and Deepak Kapur) – 1979 to 1983
- Higher-order programming in Scheme (with Aaron Kershenbaum and Dave Musser) – 1984 to 1986
- Ada Generic Library (with Dave Musser) – 1986
- UKL Standard Components – 1987 to 1988
- STL (with Meng Lee and Dave Musser) – 1993 to 1994
- SGI STL (with Matt Austern and Hans Boehm) – 1995 to 1998

Fundamental Principles

- Systematically identifying and organizing useful algorithms and data structures
- Finding the most general representations of algorithms
- Using whole-part value semantics for data structures
- Using abstractions of addresses as the interface between algorithms and data structures

Identification and Organization

1. Find algorithms and data structures

2. Implement them

3. Create usable taxonomy

Finding software components

- Books, papers
- Other libraries
- Real codes

Implementing Components

- Specify correct interfaces
- Implement
- Validate
- Measure

Organizing Components

- Fill the gaps
- Define orthogonal structure based on functionality
- Document

Generic Programming

1. Take a piece of code
2. Write specifications
3. Replace actual types with formal types
4. Derive requirements for the formal types that imply these specifications

Whole-part semantics

- Data structures extend the semantics of structures
- Copy of the whole copies the parts
- When the whole is destroyed, all the parts are destroyed
- Two things are equal when they have the same number of parts and their corresponding parts are equal

Addresses / Iterators

- Fast access to the data
- Fast equality on iterators
- Fast traversal operations – different for different categories

Iterator Categories

- Input
- Output
- Forward
- Bidirectional
- Random-access
- Two-dimensional
- …

Abstraction Mechanisms in C++

- Object Oriented Programming
  - Inheritance
  - Virtual functions
- Generic Programming
  - Overloading
  - Templates

Both use classes, but in a rather different way

Object Oriented Programming

- Separation of interface and implementation
- Late or early binding
- Slow
- Limited expressability
  - Single variable type
  - Variance only in the first position

Generic Programming

- Implementation is the interface
  - Terrible error messages
  - Syntax errors could survive for years
- Early binding only
- Could be very fast
  - But potential abstraction penalty
- Unlimited expressability

Reduction operator

```
template <class InputIterator, class BinaryOperation>
typename iterator_traits<InputIterator>::value_type
reduce(InputIterator first,
          InputIterator last,
          BinaryOperation op) {
     if (first == last) return identity_element(op);
     typename iterator_traits<InputIterator>::value_type
          result = *first;
     while (++first != last) result = op(result, *first);
     return result;
}
```

Reduction operator with a bug

```
template <class InputIterator, class BinaryOperation>
typename iterator_traits<InputIterator>::value_type
reduce(InputIterator first,
         InputIterator last,
      BinaryOperation op) {
   if (first == last) return identity_element(op);
   typename iterator_traits<InputIterator>::value_type
      result  =  *first;
   while (++first <  last) result  =  op(result, *first);
   return result;
}
```

We need to be able to define what InputIterator is in the language in which we program, not in English

Concepts

```
concept SemiRegular : Assignable, DefaultConstructible{};
concept Regular : SemiRegular, EqualityComparable {};
concept InputIterator : Regular, Incrementable {
    SemiRegular value_type;
    Integral distance_type;
    const value_type& operator*();
};
```

Reduction done with Concepts

```
value_type(InputIterator) reduce(InputIterator first,
                                 InputIterator last,
                                 BinaryOperation op)
(value_type(InputIterator) == argument_type(BinaryOperation)) {
    if (first == last) return identity_element(op);
    value_type(InputIterator) result = *first;
    while (++first != last) result = op(result, *first);
    return result;
}
```

Signature of merge

```
OutputIterator merge(InputIterator[1] first1,
                     InputIterator[1] last1,
                     InputIterator[2] first2,
                     InputIterator[2] last2,
                        OutputIterator result)
(bool operator<(value_type(InputIterator[1]),
                     value_type(InputIterator[2])),
 output_type(OutputIterator) == value_type(InputIterator[1]),
 output_type(OutputIterator) == value_type(InputIterator[2]));
```

Virtual Table for InputIterator

- type of the iterator
  - copy constructor
  - default constructor
  - destructor
  - `operator=`
  - `operator==`
  - `operator++`
- value type
- distance type
- `operator*`

Unifying OOP and GP

- Pointers to concepts
- Late or early binding
- Well defined interfaces
- Simple core language

Industrial Revolution in Software

- Large, systematic catalogs
- Validated, efficient, generic components
- Component engineers (few)
- System engineers (many)

Changes in Industry

- Industry
  - Code is a liability
  - Internal code tax
  - Continuous professional education
- Government
  - Tax support for the fundamental infrastructure
  - Legal framework
- Academia

Is STL successful?

- Millions of copies out
- Everybody (Microsoft, IBM, Sun …) ships it
- A dozen books

- Very few extensions
- No language progress
- No effect on software engineering