Generic Programming Concepts: STL and Beyond

David R. Musser
Rensselaer Polytechnic Institute
musser@cs.rpi.edu

July 10, 2002

A tutorial presented at the Program Composition for Embedded Systems Principal Investigators Meeting, Beaverton, Oregon, July 10-12, 2002.

Click here for printer-friendly version

In this PDF screen version of the slides, links to other supplementary slides are colored red. These additional slides are in html format, generated from Microsoft Powerpoint format; they have not been converted into PDF for direct incorporation in this document since they contain a few animations of diagrams. The best way to use these links depends on whether you view this PDF file in a Web browser or download it and view it in Adobe Acrobat.

- If you are using a Web browser, you can left-click the links as usual and remain within the same browser; however, when you are finished with a set of supplementary slides, the Back button on the browser will inconveniently take you back to the beginning of the PDF file. Thus you may instead want to right-click the link, select "Copy Link Location" from the pop-up menu, and paste it into the Location pane of a separate browser window. Then when you finish with a set of supplementary slides you can bring up original browser window and continue the main slides from the point where you left off.

- If you are using Adobe Acrobat, when you encounter the links to supplementary slides you should *not* simply left-click on them, as Acrobat will try (unsuccessfully) to try to display them with its own very limited Web-browsing capability. Instead, right-click and select "Open Weblink in Browser" from the pop-up menu to begin viewing the supplementary slides in a separate Web browser window. When finished with those supplementary slides, bring up the Acrobat window again to continue with the main slides.

In case your browser has difficulty displaying the animations in the html format, you may prefer to download and view the original Powerpoint versions of the supplementary slides from

`http://www.cs.rpi.edu/~musser/gp/GPtutorial`.

Contents

1. Generic Programming: First Definition, and STL Examples

Exploring methods of developing, organizing and using libraries of reusable software components

Distinguished from other work on software reuse by emphasis on

- high degree of adaptability
- requirement that the components be *efficient*

Example: Standard Template Library portion of the C++ Standard Library

1.1. *Standard* Template Library

- The C++ Standard Template Library (STL) was proposed to the ANSI/ISO C++ Standards Committee in 1994 and, after small revisions and extensions, became part of the official C++ Standard in 1997

- Standardization of previously designed and implemented generic libraries in C++ (Stepanov, Lee, Musser)

- Standardization has many benefits, including the fact that it is a *useful constraint* on design and implementation choices

1.1.1. Performance requirements as part of the standard

- Make time and space bounds *part of the contract* for data abstractions and algorithms

- How to express the bounds for *generic* algorithms?

- Starting point: Asymptotic formulas (O, Θ, Ω bounds)

- Greater precision required in some cases, e.g., to distinguish two sorting algorithms that both have $O(N \log N)$ bounds

- Is there something better? . . . more on this later

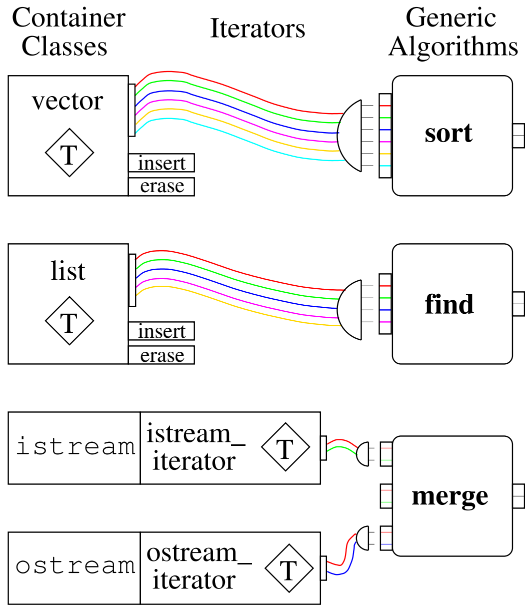
1.2. Standard *Template* Library

- Two major approaches to polymorphism in programming:
  - *subtype polymorphism*, realized with inheritance and virtual functions: interface requirements are written as virtual functions of an abstract base class, from which concrete classes are derived
  - *parametric polymorphism*, realized with generics or templates that specify a class or function with type parameters
- C++ supports both
  - subtype polymorphism, with class inheritance and dynamic binding of virtual member functions, and
  - parametric polymorphism, with templates, static typing, function overloading, and function inlining
- STL uses only parametric polymorphism—but generic programming is *not* limited to parametric polymorphism

1.3. Standard Template *Library*

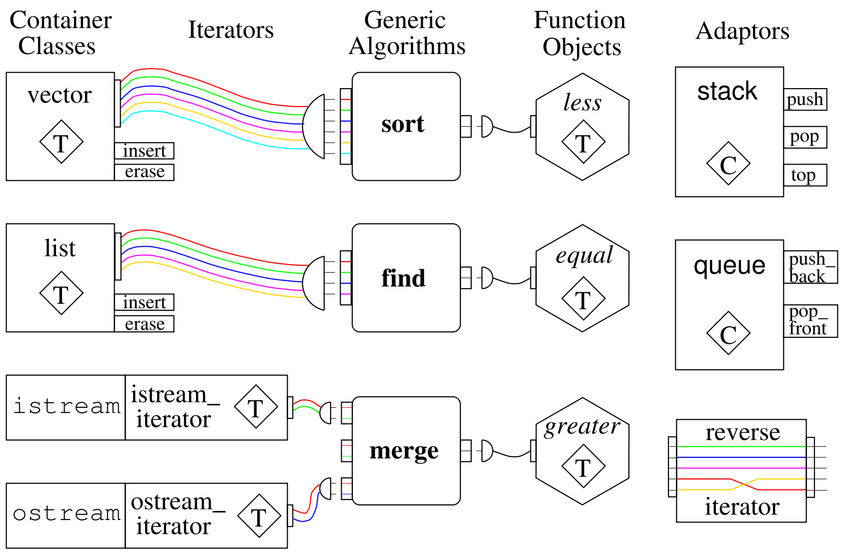
- STL is a general-purpose library of generic algorithms and data structures communicating through iterators
- Contains many small components that can be plugged together and used in an application
- Functionality and performance requirements are carefully chosen so that there are a myriad of useful combinations

Connecting Containers and Algorithms with Iterators



++	***Increment***	==, &	***Compare, Reference***
=	***Assign***	--	*Decrement*
*	***Dereference***	+, -, <	*Random Access*

Five of the Six Major STL Component Categories (Concepts)



Container Classes

Iterators

Generic Algorithms

Function Objects

Adaptors

vector T insert erase

sort

less T

stack C push pop top

list T insert erase

find

equal T

queue C push_ back pop_ front

`istream` istream_ iterator T

merge

greater T

reverse iterator

`ostream` ostream_ iterator T

`++` ***Increment***

`=` ***Assign***

`*` ***Dereference***

`==, &` ***Compare, Reference***

`--` *Decrement*

`+, -, <` *Random Access*

◇ *Generic Parameter*

1.4. C++ Features

- (Almost) strongly typed: contractual model
- Supports modern software engineering methods:
    - Abstract Data Types (modular programming, information hiding)
    - Object-Oriented Programming (classes, inheritance, subtype ¡polymorphism, late binding)
    - Generic Programming

1.4.1. C++ Features most relevant to STL

- Generic programming support
  - Template classes and functions, partial specialization
  - Function inlining
  - Operator overloading
- Systems programming support
  - Complete control of low-level details, when needed
- Object oriented programming support
  - Classes, inheritance
  - Automatically invoked constructors and destructors
  - *Not used:* late-binding with virtual functions

1.5. Container example: application of STL map

A “terminal manager,” based on a simple example of embedded systems code [2]:

```
class TerminalManager {
   map<int, Terminal *> terminalMap;
public:
  Status AddTerminal(int terminalId, int type) {
    Status status;
    if (terminalMap.count(terminalId) == 0) {
      Terminal* pTerm = new Terminal(terminalId, type);
      terminalMap[terminalId] = pTerm;
      status = SUCCESS;
    } else
      status = FAILURE;
    return status;
  }
```

```
Status RemoveTerminal(int terminalId) {
  Status status;
  if (terminalMap.count(terminalId) == 1) {
    Terminal* pTerm = terminalMap[terminalId];
    terminalMap.erase(terminalId);
    delete pTerm;
    status = SUCCESS;
  } else
    status = FAILURE;
  return status;
}
Terminal* FindTerminal(int terminalId) {
  Terminal* pTerm;
  if (terminalMap.count(terminalId) == 1) {
    pTerm = terminalMap[terminalId];
  } else
    pTerm = NULL;
  return pTerm;
}
```

```
  void HandleMessage(const Message* pMsg) {
    int terminalId = pMsg->GetTerminalId();
    Terminal* pTerm  = FindTerminal(terminalId);
    if (pTerm)
      pTerm->HandleMessage(pMsg);
  }
}; // end of class TerminalManager

int main()
{
  TerminalManager TH;
  TH.AddTerminal(700000013, 1);
  TH.AddTerminal(700000017, 2);
  TH.AddTerminal(700000007, 1);
  // ...
  Terminal* t = TH.FindTerminal(700000017);
  // ...
  TH.RemoveTerminal(700000017);
  // ...
}
```

1.6. Generic algorithm example: application of generic merge

```
#include <cassert>
#include <list>
#include <deque>
#include <algorithm>  // For merge
using namespace std;

template <typename Container>
Container make(const char s[])
{
  return Container(&s[0], &s[strlen(s)]);
}
```

continued on next slide . . .

```
int main()
{
  cout << "Demonstrating generic merge algorithm with "
       << "an array, a list, and a deque." << endl;
  char s[] = "aeiou";
  int len = strlen(s);
  list<char> list1 =
     make< list<char> >("bcdfghjklmnpqrstvwxyz");

  // Initialize deque1 with 26 copies of the letter x:
  deque<char> deque1(26, ’x’);

  // Merge array s and list1, putting result in deque1:
  merge(&s[0], &s[len], list1.begin(), list1.end(),
        deque1.begin());
  assert (deque1 ==
          make< deque<char> >("abcdefghijklmnopqrstuvwxyz"));
}
```

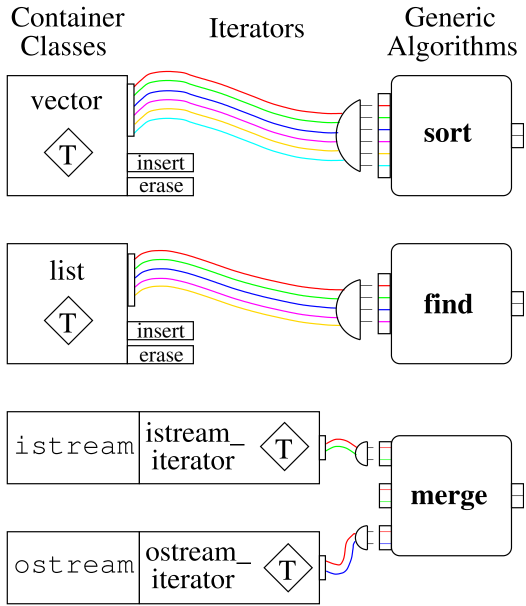
1.7. The central role of iterators

Iterators: pointer-like objects used by algorithms to traverse sequences of objects stored in a container

“Pointers on steroids”

STL generic algorithms like `merge` are written in terms of iterator parameters, and STL containers provide iterators that can be plugged into the algorithms

Again: Connecting Containers and Algorithms with Iterators



++	***Increment***	==, &	***Compare, Reference***
=	***Assign***	--	*Decrement*
*	***Dereference***	+, -, <	*Random Access*

1.8. Another generic algorithm example: accumulate

```
#include <iostream>
#include <vector>
#include <cassert>
#include <numeric>  // for accumulate
using namespace std;

int main()
{
  int x[5] = {2, 3, 5, 7, 11};

  // Initialize vector1 to x[0] through x[4]:
  vector<int> vector1(&x[0], &x[5]);

  int sum = accumulate(vector1.begin(), vector1.end(), 0);

  assert (sum == 28);
}
```

1.9. Definition of accumulate: how it places certain requirements on iterators

```
template <typename InputIterator, typename T>
T accumulate(InputIterator first, InputIterator last, T init)
{
  while (first != last) {
    init = init + *first;
    ++first;
  }
  return init;
}
```

1.10. Trait classes and partial specialization

```
template <typename Iterator>
struct iterator_traits {
 typedef typename Iterator::value_type value_type;
 typedef typename Iterator::difference_type difference_type;
 typedef typename Iterator::pointer pointer;
 typedef typename Iterator::reference reference;
 typedef typename Iterator::iterator_category
      iterator_category;
};
template <typename T>
struct iterator_traits<T*> {
 typedef T value_type;
 typedef ptrdiff_t difference_type;
 typedef T* pointer;
 typedef T& reference;
 typedef random_access_iterator_tag iterator_category;
};
```

1.11. Compile-time algorithm dispatching

```
template <typename ForwardIterator>
void  __rotate(ForwardIterator first, ForwardIterator middle,
               ForwardIterator last,  forward_iterator_tag);

template <typename BidirIterator>
void
void __rotate(BidirIterator first, BidirIterator middle,
            BidirIterator last,  bidirectional_iterator_tag);

template <typename RanAccIterator>
void __rotate(RanAccIterator first, RanAccIterator middle,
           RanAccIterator last,  random_access_iterator_tag);

template <typename Iterator>
inline void rotate (Iterator first, Iterator middle,
                    Iterator last)
{ __rotate(first, middle, last,
           iterator_traits<Iterator>::iterator_category());}
```

1.12. Function objects

Function objects (a.k.a. functors) are objects that can be applied to arguments to produce an effect and/or a value

Can be used to make algorithms like `accumulate` more generic:

```
template <typename InputIterator, typename T,
          typename BinaryOperation>
T accumulate(InputIterator first, InputIterator last,
             T init, BinaryOperation binary_op)
{
  while (first != last) {
    init = binary_op(init, *first);
    ++first;
  }
  return init;
}
```

1.12.1. Accumulate with mult as binary op to produce product of sequence elements

```
int mult(int x, int y) { return x * y; }

int main()
{
  int x[5] = {2, 3, 5, 7, 11};
  // Initialize vector1 to x[0] through x[4]:
  vector<int> vector1(&x[0], &x[5]);
  int product = accumulate(vector1.begin(), vector1.end(),
                           1, mult);
  assert (product == 2310);
}
```

1.12.2. Another way to define the mult function object

```
struct multiplies {
  int operator() const (int x, int y) { return x * y; }
} mult;

int main()  // same as before
{
  int x[5] = {2, 3, 5, 7, 11};
  // Initialize vector1 to x[0] through x[4]:
  vector<int> vector1(&x[0], &x[5]);
  int product = accumulate(vector1.begin(), vector1.end(),
                           1, mult);
  assert (product == 2310);
}
```

Advantages include better optimization and ability of the functor to carry additional information such as number and types of its arguments, or even to carry state in member variables.

1.13. Adaptors

Adaptors: components that are applied to other components to produce

- *a restricted interface*, e.g.,

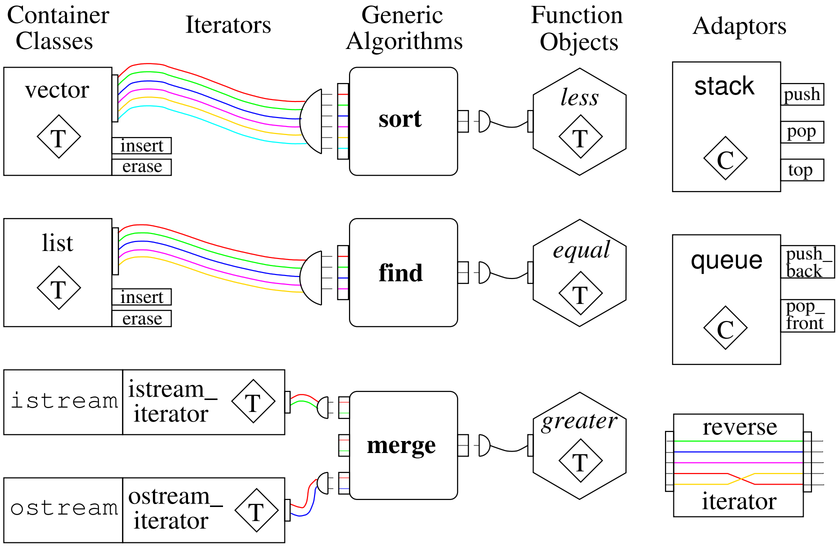
  ```
  template <typename T, typename Container = deque<T> >
  class stack;
  ```

  Others in STL are `queue`, `priority_queue`

- *modified functionality*, e.g, `reverse_iterator`

- *extended functionality*, e.g, `counting_iterator` (not in STL; does what base iterator does but also keeps track of operation counts)

Adaptors include container adaptors and iterator adaptors



++	*Increment*	==, &	*Compare, Reference*	◇	*Generic Parameter*
=	*Assign*	--	*Decrement*		
*	*Dereference*	+, -, <	*Random Access*		

1.14. Accumulate with a reverse iterator

```
//... headers and namespace declaration
int main()
{
  cout << "Demonstrating generic accumulate algorithm with "
       << "a reverse iterator." << endl;
  float small = (float)1.0/(1 << 26);
  float x[5] = {1.0, 3*small, 2*small, small, small};

  // Initialize vector1 to x[0] through x[4]:
  vector<float> vector1(&x[0], &x[5]);

  cout << "Values to be added: " << endl;

  vector<float>::iterator i;
  cout.precision(10);
  for (i = vector1.begin(); i != vector1.end(); ++i)
    cout << *i << " ";
  cout << endl;
  float sum = accumulate(vector1.begin(), vector1.end(),
                         (float)0.0);

  cout << "Sum accumulated from left = " <<  sum << endl;

  float sum1 = accumulate(vector1.rbegin(), vector1.rend(),
                          (float)0.0);

  cout << "Sum accumulated from right = "
       << (double)sum1 << endl;
}
```

Output:

```
Values to be added:
1 4.470348358e-08 2.980232239e-08 1.490116119e-08 1.490116119e
Sum accumulated from left = 1
Sum accumulated from right = 1.00000119
```

1.15. TerminalManager generalized and rewritten as a container adaptor

```
template <typename Resource, typename ResourceId,
  typename ResourceType, typename Object,
  typename UniqueAssociativeContainer
             = map<ResourceId, Resource* > >
class ResourceManager {
   UniqueAssociativeContainer resourceMap;
public:
  Status AddResource(int resourceId, int type) {
    Status status;
    if (resourceMap.count(resourceId) == 0) {
      Resource* q = new Resource(resourceId, type);
      resourceMap[resourceId] = q;
      status = SUCCESS;
    } else
      status = FAILURE;
    return status;
  }
```

```
Status RemoveResource(int resourceId) {
  Status status;
  if (resourceMap.count(resourceId) == 1) {
    Resource* q = resourceMap[resourceId];
    resourceMap.erase(resourceId);
    delete q;
    status = SUCCESS;
  } else
    status = FAILURE;
  return status;
}
Resource* FindResource(int resourceId) {
  Resource* q;
  if (resourceMap.count(resourceId) == 1) {
    q = resourceMap[resourceId];
  } else
    q = NULL;
  return q;
}
void HandleObject(const Object* p) {
  int resourceId = p->GetResourceId();
  Resource* q  = FindResource(resourceId);
```

```
    if (q)
      q->HandleObject(p);
  }
};
```

1.16. Some instances of ResourceManager

```
// Following is same as TerminalManager
ResourceManager<Terminal, int, int, Message> TM1;

// Following is same except uses a hash_map
ResourceManager<Terminal, int, int, Message,
                hash_map<int, Terminal* > > TM2;
```

2. Generic Programming: A Broader Definition

Generic Programming = Programming with Concepts

What this means . . .

3. BGL (the Boost Graph Library)

- Graph algorithms typically need great flexibility in traversing and visiting vertices and edges of a graph.

- BGL, the Boost Graph Library,[1] achieves the requisite flexibility through advanced generic programming techniques, including

  - provision for several kinds of iteration through vertices, edges, adjacent vertices, etc.
  - provision for selectively applying several different operations on each visited vertex or edge, as is commonly needed in graph algorithms, via a novel Visitor Concept (which extends the Functor Concept): Visitor Concepts.

- Some examples: Quick Tour and Example Programs

[1] Developed by Andrew Lumsdaine's research group (formerly at the University of Notre Dame, now at Indiana University).

4. New Ways of Specifying and Using Performance Requirements

A fundamental problem in standardizing software component libraries is, how can the library standard be written so that

- library implementors have the freedom to take some advantage of hardware/OS/compiler environment characteristics, yet
- application programmers have sufficient guarantees about the performance of library components that they can easily port their programs from one environment to another.

A possible solution is *algorithm concept hierarchies*, which may also be the best means to give compiler optimizations access to sufficiently accurate characterizations of algorithms.

4.1. Performance guarantees in the C++ standard

- The requirements are mostly stated in traditional O-notation, which suppresses constant factors and thus is incapable of distinguishing between two algorithms with the same asymptotic behavior but different constant factors.

- In some cases exact or approximate bounds are given for operation counts, of some principal operation (like comparison operations, in sorting algorithm descriptions).

4.2. A typical algorithm specification in the C++ standard

heapsort (Simplified from *partial_sort*'s description.)

Prototype:

```
template <typename RandomAccessIterator>
void heapsort(RandomAccessIterator first,
              RandomAccessIterator last)
```

Effects: Sorts the elements in the range [first, last), in place.

Complexity: It takes approximately $N \log N$ comparisons, where $N = \text{last} - \text{first}$.

4.3. How can we characterize performance more precisely?

- Actual times, machine instruction counts or memory accesses are too specific—nonportable—even for nongeneric components

- Counting only one operation, like comparisons, ignores variation that can occur when, say, the algorithm is instantiated with a more expensive iterator type

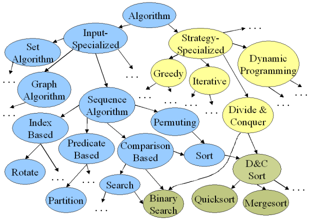
- E.g., for generic sorting algorithms we need to count

  - value comparisons
  - value assignments
  - iterator operations
  - "distance-type" operations

4.4. Algorithm concepts for setting performance standards

- Develop *algorithm concept hierarchies* similar to previously developed hierarchies for container and iterator concepts.

  Click on the diagram to enlarge it and browse the hierarchy.



- Use these algorithm concepts to present and organize performance requirements for a standard library's algorithm components.

- Start with performance requirements expressed in terms of operation counts of *all* operations that are introduced through type or functor parameters.

- Extend it so that it takes into account key hardware characteristics such as cache size and speed.

4.5. Why principal operation counting is not enough

- Looking only at principal operation counts ignores important hardware differences, in

  > available instruction sets, number and speed of arithmetic units, registers, size and speed of caches and memory, etc.

- This abstractness can lead to suboptimal choices of algorithms for a particular task.

4.6. Extending principal operation counting

- How we might extend principal operation counting to take better account of hardware differences:
  - Express algorithms with additional parameters that *capture key hardware characteristics as a concept*, e.g., a cache concept.
  - Then study the performance of different algorithms or algorithm variants as assumptions about these parameters are varied—i.e., *are refined into different subconcepts*.

4.7. Organizing details and summarizing statistics

- Main drawback to introducing and varying hardware (and OS and compiler) parameters: the *amount of detail* that must be reported to give a fully accurate picture of an algorithm's performance.

- But organization of information using concept hierarchies could help in

  - *suppression of details* at one level while revealing them fully at deeper levels;
  - summarization, aggregation of statistics.

- *providing a database of detailed performance statistics*, corresponding to different environments, that can be accessed at both compile time and run time:
  - to help make the most appropriate choice of algorithms from a library depending on the specific environment in which an computation is to be performed, and thus
  - to assist overall in optimizing applications for a particular environment

5. It's Not Just Libraries

In the Simplicissimus Project, generic programming is playing a major role in developing better compiler optimizations, by raising the abstraction level at which optimizations can occur.

Simplicissimus is a major component of an NSF Next Generation Software project, "Open Compilation for Self-Optimizing Generic Component Libraries" [Sibylle Schupp, David Musser, Douglas Gregor, Brian Osman (Rensselaer Polytechnic Institute); Andrew Lumsdaine, Jeremy Siek, Lie-Quan Lee (Indiana University); industry collaborator: S.-M. Liu (HP, formerly SGI)]

6. Generic Programming in Java

- Some libraries developed with the existing language
  - Java Generic Library (JGL), by ObjectSpace
  - Java Algorithms Library (JAL), by SGI
- What's happening with templates in Java?
  - Generic Java (GJ, formerly Pizza), by Bracha, Odersky, Stoutamire, and Wadler—implements class templates via type erasure
  - NextGen, by Cartwright and Steele—removes some of the restrictions imposed by GJ, like lack of runtime genericity
  - Sun JSR-014, based on GJ, seems to ignore NextGen
  - None addresses extensions needed to fully compete with C++, like function templates, partial specialization, or even operator overloading

7. Generic Embedded Systems Programming

- Today, there are some benefits of using existing generic libraries

- E.g., STL containers and container adaptors have proved useful in implementing embedded systems design patterns, such as

  - message queues, with STL `queue` and `priority_queue` adaptors
  - resource allocators, with STL `stack` and `queue` adaptors
  - resource managers and routing tables, with the STL `map` container.
  - publish-subscribe event handlers, with STL `vector` or `list` containers

- BGL graph containers and algorithms have many applications to network design patterns such as distance-vector or link-state routing, easily programmable using BGL's highly adaptable Bellman-Ford or Dijkstra shortest paths algorithms.

- BGL's graph-related concepts may also be a useful basis for extensions to graph-concept-based distributed computing algorithms such as breadth-first-search approaches to broadcast communication, leader election, etc.

- Going further . . .

8. How Does Generic Programming Relate to Aspect Oriented Programming?

- Recall the definition

  Generic Programming = Programming with Concepts.

  Or, as Alex Stepanov has said, "The goal of Generic Programming is to provide a systematic classification of useful software components ... Generic Programming is fundamentally a study of algorithms, data structures and other software components with particular emphasis on their systematic organization."

- Thus, it is largely *inappropriate* to compare GP directly with programming paradigms such as Object Oriented Programming or Functional Programming, where one's attention is focused more on the benefits of a particular way of designing and implementing applications (as opposed to components).

- Aspect Oriented Programming, on the other hand, does emphasize the importance of conceptual organization, in terms of aspects:
  - whereas STL components are usually described as "fine-grained" in comparison to previous software components, aspects are even finer-grained, allowing greater control of variations;
  - with these finer-grained aspects, though, comes an even stronger need for conceptual organization.
- Thus we can hope that Aspect Oriented Programming can derive important benefits from what has been learned about conceptual organization from STL and more recent results of Generic Programming.

References

[1] Matt Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1999.

[2] EventHelix, Inc. STL design patterns. http://www.eventhelix.com.

[3] Douglas P. Gregor, Sibylle Schupp, and David R. Musser. Design patterns for library optimizations. In J.Strieglitz K. Davis, editor, *Proc. International Conf. on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'01) Tampa, FL*, 2001.

[4] International Organization for Standardization (ISO). *ISO/IEC Final Draft International Standard 14882: Programming Language C++*. 1 rue de Varembé, Case postale 56, CH-1211 Genève 20, Switzerland, 1998.

[5] Y. Ishikawa et al. MPC++. In G. Wilson and P. Lu, editors, *Parallel Programming Using C++*, pages 427–466. MIT Press, 1996.

[6] Donald E. Knuth. *The Art of Computer Programming, Vols. 1-3*. Addison Wesley, 1998.

[7] Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. The generic graph component library. In *Proceedings OOPSLA'99*, 1999.

[8] David R. Musser and Alexander A. Stepanov. Algorithm-oriented generic libraries. *Software–Practice and Experience*, 24(7):623–642, July 1994.

[9] David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Second Edition*. Addison-Wesley, Reading, MA, 550 pages, 2001.

[10] Sibylle Schupp, Douglas P. Gregor, David R. Musser, and Shin-Ming Liu. Library transformations. In Mark Hamann,

editor, *Proc. IEEE Internat. Conf. Source Code Analysis and Manipulation, Florence*, pages 109–121, 2001.

[11] Sibylle Schupp, Douglas P. Gregor, David R. Musser, and Shin-Ming Liu. User-extensible simplification—type-based optimizer generators. In Reinhard Wilhelm, editor, *International Conference on Compiler Construction*, Lecture Notes in Computer Science, 2001.

[12] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library User Guide and Reference Manual*. Addison Wesley, 2002.

[13] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

[14] Silicon Graphics Inc. Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl, 1997.

[15] Rudolf Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. In I. Rival, editor, *Ordered Sets*, pages 445–470. Reidel, Dordrecht-Boston, 1982.

[16] Rudolf Wille. Concept lattices and conceptual knowledge systems. *Computers and Mathematics with Applications*, 23:493–522, 1992.