Algorithm Library Design: Crash Course in Modern C++

Based on the
Algorithm Library Design Course 2003
http://www.mpi-sb.mpg.de/~kettner/courses/lib_design_03/

Lutz Kettner
MPI für Informatik, Saarbrücken

April 5, 2005
Algorithm and Data Structure Libraries

• LEDA – Library of Efficient Datatypes and Algorithms
  ▪ C++, uses templates for container, static graphs, …

• STL – Standard Template Library
  ▪ C++ standard library, almost exclusively template code

• BOOST – collection of various C++ libraries
  ▪ BGL – Boost Graph Library

• CGAL – Computational Geometry Algorithms Library
  ▪ C++, almost exclusively template code, follows STL design

• EXACUS – Efficient and Exact Alg. for Curves and Surfaces
  ▪ C++, almost exclusively template code, follows CGAL design
Contents

• C++ Templates
• Generic Programming
Function Template

template <class T>
void swap( T& a, T& b) {
    T tmp = a; a = b; b = tmp;
}

int main() {
    int i = 5; int j = 7;
    swap( i, j); // uses "int" for T.
}
Class Template

template <class T>
struct vector {
    void push_back( const T& t); // append t to vector.
};

int main() {
    vector<int> vs; // uses "int" for T.
    vs.push_back(5);
}
Class Template

template <class T>
struct vector {
    void push_back( const T& t); // append t to vector.
};

template <class T>
void vector<T>::push_back( const T& t) { ... }

int main() {
    vector<int> vs; // uses "int" for T.
    vs.push_back(5);
}
Advantages of Templates

• Static polymorphism
  - flexibility (but at compile time)

• Strong type checking
  - compile-time instantiation allows for type checking not possible with the conventional runtime polymorphism of object orientation or function pointers

```c
void qsort( void *base, size_t nel, size_t width,
            int (*compare)( const void *, const void *));
```

• Efficient inlining and static optimizations

```c
template <class Compare>
void qsort( void *base, size_t nel, size_t width, Compare cmp);
```
Pattern Matching in Function Templates

template <class T>
void foo(vector<T>& vs);
Default Template Arguments

template <class T, class Alloc = std::allocator<T> >

struct vector { ... };
Integral Built-in Types as Template Arguments

template <int dim>
struct Point {
    double coordinates[dim]; // coordinate array
    // ...
};

int main() {
    Point<3> // a point in 3d space
}

Member Templates

template <class T1, class T2>
struct pair {
    T1 first;
    T2 second;
    pair() {} // explicit default constructor
    template <class U1, class U2> // template constructor
    pair( const pair<U1,U2>& p);
};
Member Templates

template <class T1, class T2>
struct pair {
    T1 first;
    T2 second;
    pair() {}  // explicit default constructor
    template <class U1, class U2>  // template constructor
    pair( const pair<U1,U2>& p); 
};
template <class T1, class T2>
template <class U1, class U2>
pair<T1,T2>::pair( const pair<U1,U2>& p)
    : first( p.first), second( p.second) {}
Template Full Specialization

template <class T>
struct vector;

template <>
struct vector<bool> {
    // specialized implementation
};
Template Partial Specialization

template <class T, class Alloc = std::allocator>
struct vector;

template <class Alloc>
struct vector<bool, Alloc> {
    // specialized implementation
};
Member Types

template <class T>
struct vector {
    typedef T value_type;
};

int main() {
    list<int> ls;
    list<int>::value_type i; // is of type int
}
Member Types

template <class T>
struct vector {
    typedef T value_type;
};

template <class Container>
struct X {
    typedef Container::value_type value_type; // not correct
    // ...
};
Member Types: typename keyword

template <class T>
struct vector {
    typedef T value_type;
};

template <class Container>
struct X {
    typedef typename Container::value_type value_type;
    // ...
};
Templates are Turing-computable

- Enums as integers
- Partial template specialization for if’s
  - template <int condition, class ThenType, class ElseType>
    struct MetaIf {
    typedef ThenType Result;
    
    template <class ThenType, class ElseType>
    struct MetaIf<0,ThenType,ElseType> {
    typedef ElseType Result;
    
    template <int condition, class ThenType, class ElseType>
    struct MetaIf {
    typedef ThenType Result;
    
    template <class ThenType, class ElseType>
    struct MetaIf<0,ThenType,ElseType> {
    typedef ElseType Result;
    
- Recursive and partial template instantiations for recursion
// Program by Erwin Unruh, adapted to C++ standard.
// Compile with: g++ -ftemplate-depth-50 -c prime.C | & grep conversion

template <int i, int prim> struct D {} ;    // "output" at compile time (error messages)
template <int i> struct D<i,0> { D(i);};

template <int p, int i> struct is_prime {
    // compute prime condition
    enum { prim = ((p%i) && is_prime<i>(i>2 ? p : 0), i-1>::prim) };
};
template<> struct is_prime<0,1> { enum { prim = 1}; };
template<> struct is_prime<0,0> { enum { prim = 1}; }

template <int i> struct Prime_print {
    // iterate through all values: 2..i
    Prime_print<i-1> a;
    enum { prim = is_prime<i,i-1>::prim };  
    void f() { a.f(); D<i,prim> d = prim; }
};
template<> struct Prime_print<2> {
    enum { prim = 1};
    void f() { D<2,prim> d = prim; }
};

void foo() { Prime_print<17> a; a.f(); }
Standard Template Library STL

- Part of the ISO/OSI C++ Standard Library
- Provides basic data types, such as `list`, `vector`, `set`, `map`, …
- and basic algorithms, such as `find`, `sort`, …
- Good online reference manuals http://www.sgi.com/tech/stl/
- Prime example of the generic programming paradigm
Generic Programming Paradigm

[Jazaeri98]:

- **Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction.**
Generic Programming Paradigm

[Jazaeri98]:

- *Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction.*

**Key ideas include:**

- *Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.*
Generic Programming Paradigm

[Jazaeri98]:

- Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction.

Key ideas include:

- Lifting of a concrete algorithm to as general a level as possible without losing efficiency; i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.
Generic Programming Paradigm

[Jazaeri98]:

• **Generic programming** is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction.

**Key ideas include:**

• When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.
Generic Programming Paradigm

[Jazaeri98]:

- Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction.

Key ideas include:

- Providing more than one generic algorithm for the same purpose and at the same level of abstraction, when none dominates the others in efficiency for all inputs. This introduces the necessity to provide sufficiently precise characterizations of the domain for which each algorithm is the most efficient.
Generic Programming Paradigm

Summary:

• Focus on algorithms
• Focus on efficiency
• Focus on abstraction of assumptions about data
**Example: remove_if_divides**

A short program in [Stepanov&Lee 95] makes us if this negator. The program copies all integers from cin to cout that cannot be divided by the integer parameter given to the program.

```cpp
int main( int argc, char** argv) {
    if ( argc != 2)
        throw( "usage: remove_if_divides integer\n");
    remove_copy_if( istream_iterator<int>(cin), istream_iterator<int>(),
                    ostream_iterator<int>(cout, "\n"),
                    not1( bind2nd( modulus<int>(), atoi( argv[1]))));
    return 0;
}
```
Concept and Model

template <class T>
void swap( T& a, T& b) { T tmp = a; a = b; b = tmp; }

• T requires a default constructor and assignment
• Collections of requirements are called concepts
  ▪ Syntactic requirements, e.g., existence of operators and functions
  ▪ Semantic requirements, e.g., behavior or runtime complexity
• Concepts are used to document template parameters, e.g.,
  T requires the Assignable concept.
• If an actual type fulfills the requirements of a concept, it is a called a model for this concept, e.g.,
  int is a model of the Assignable concept.
# Common Basic Concepts

<table>
<thead>
<tr>
<th>Concept</th>
<th>Syntactic requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignable</td>
<td>copy constructor, assignment operator</td>
</tr>
<tr>
<td>Default Constructible</td>
<td>default constructor</td>
</tr>
<tr>
<td>Equality Comparable</td>
<td>equality and inequality operator</td>
</tr>
<tr>
<td>LessThan Comparable</td>
<td>order comparison with operators <code>&lt;</code>, <code>&lt;=</code>, <code>&gt;=</code>, and <code>&gt;</code></td>
</tr>
</tbody>
</table>
Data Structures and Algorithms

Data Structures
- vector
- list
- deque
- map
- set
- ...

Algorithms
- copy
- find
- transform
- sort
- generate
- nth_element
- ...

O(n^2)-combinations
Generic Algorithms Based on Iterators

Data Structures
- vector
- list
- deque
- map
- set
- ...

Iterators

Algorithms
- copy
- find
- transform
- sort
- generate
- nth_element
- ...

O(n)-combinations
## Iterator Concept Refinement Hierarchy

<table>
<thead>
<tr>
<th>Concept</th>
<th>Refinement of</th>
<th>Syntactic requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trivial Iterator</td>
<td>Assignable, Equality Comparable</td>
<td>operator*( )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>operator-&gt;()</td>
</tr>
<tr>
<td>Input Iterator</td>
<td>Trivial Iterator</td>
<td>operator++(), ...</td>
</tr>
<tr>
<td>Output Iterator</td>
<td>Assignable</td>
<td>operator*( ), operator++() ...</td>
</tr>
<tr>
<td>Forward Iterator</td>
<td>Input Iterator, Output Iterator, Default Constructible</td>
<td>...</td>
</tr>
<tr>
<td>Bidirectional Iterator</td>
<td>Forward Iterator</td>
<td>operator--(), ...</td>
</tr>
<tr>
<td>Random Access Iterator</td>
<td>Bidirectional Iterator, LessThan Comparable</td>
<td>operator+(), operator+=(), operator()-(), operator<a href=""></a> ...</td>
</tr>
</tbody>
</table>
Models of Iterators

• C pointers into C arrays are models of iterators.
• Classes can be programmed to fulfill all iterator requirements, e.g., with operator overloading, so a pointer to a linked list can be encapsulated in a class to behave like an iterator.
Iterator Ranges

Container (linear sequence)

- Iterator (begin)
- Iterator (past-the-end)
Container Provide Iterators

```
template <class T>
class list {
    void push_back( const T& t); // append t to list.
typedef UnknownInternalType iterator;
    iterator begin();
    iterator end();
};
```
template <class InputIterator, class T>
bool contains(InputIterator first, InputIterator beyond;
const T& value) {
    const T& value)
    while ((first != beyond) && (*first != value)) {
        ++first;
    }
    return (first != beyond);
}

Returns true if the value is contained in the values of the range [first, beyond).
Generic Algorithm on C Pointer

template <class InputIterator, class T>
bool contains( InputIterator first, InputIterator beyond, 
  const T& value) {
    while ((first != beyond) && (*first != value))
      ++first;
    return (first != beyond);
}

int a[100];
// ... initialize elements of a.
bool found = contains( a, a+100, 42);
Generic Algorithm on C Pointer

template <class InputIterator, class T>
bool contains( InputIterator first, InputIterator beyond, 
const T& value) {
    while ((first != beyond) && (*first != value))
        ++first;
    return (first != beyond);
}

int a[100];
// ... initialize elements of a.
bool found = contains( a, a+100, 42);
bool in_first_half = contains( a, a+50, 42);
Generic Algorithm on Container Class

template <class InputIterator, class T>
bool contains( InputIterator first, InputIterator beyond,
const T& value) {
    while ((first != beyond) && (*first != value))
        ++first;
    return (first != beyond);
}

list<int> ls;
// ... insert some elements into ls.
bool found = contains( ls.begin(), ls.end(), 42);
Generic Copy Function

template <class InputIterator, class OutputIterator>
OutputIterator
copy( InputIterator first, InputIterator beyond,
      OutputIterator result)
{
    while (first != beyond)
      *result++ = *first++;
    return result;
}
Copy Elements in int-Arrays

```c
int a1[100];
int a2[100];
// ... initialize elements of a1.
copy(a1, a1+100, a2);
```
Copy Elements in Lists

list<int> ls1;
list<int> ls2;

// ... initialize elements of ls1.
copy( ls1.begin(), ls1.end(), ls2.begin());
Copy Elements in Lists

```cpp
list<int> ls1;
list<int> ls2;

// ... initialize elements of ls1.
copy(ls1.begin(), ls1.end(), ls2.begin());

// ... writes into non-existing memory of empty list ls2!!!
```
Copy Elements in Lists

```cpp
list<int> ls1;
list<int> ls2;

// ... initialize elements of ls1.
// ... writes into non-existing memory of empty list ls2!!

copy( ls1.begin(), ls1.end(), ls2.begin() );

// ... use a small adaptor class to convert iterator calls
// ... write operations to push_back calls on container

copy( ls1.begin(), ls1.end(), back_inserter(ls2) );
```
Adaptors to Convert Between Concepts

copy( istream_iterator<int>(cin), istream_iterator<int>(),
ostream_iterator<int>(cout, "\n").

The concepts in the STL and the adaptors form an extremely flexible toolkit. Most adaptors are small classes and function. Own adaptors for other concepts are easy to add. The whole is more than the sum of its parts.
Iterator Traits

• Assume an algorithm needs the value type of an iterator
  template <class Iterator>
  void sort( Iterator first, Iterator last) {
      TYPE pivot = *first;  // ...... TYPE becomes Iterator::value_type
  }

• We could provide the type as member type of iterators
  struct iterator_over_ints {
      typedef int value_type;
      // ...
  };

• Doesn’t work with C pointers as iterators.

• We could specialize the algorithm to work with pointers:
  template <class T>
  void sort( T* first, T* last) { // ......

• Needs to be done for each algorithm ➔ factor this into own class.
Iterator Traits

• Get value type from intermediate iterator traits class
  template <class Iterator>
  void sort( Iterator first, Iterator last) {
      typename iterator_traits<Iterator>::value_type pivot = *first; // ……
  }

• In general, iterator traits get the value type from the iterator
  template <class Iterator>
  struct iterator_traits {
      typedef typename Iterator::value_type value_type;
  }

• Now we can specialize the iterator traits instead of the algorithm
  to handle the special case of pointers (and const pointers):
  template <class T>
  struct iterator_traits<T*> {
      typedef T value_type;
  }
Function Objects

(a.k.a. Functors)

A function object basically is an instance of a class with the operator() member function implemented, such that a call to this member function of the object looks like a function call.

```cpp
template <class T>
struct equals {
    bool operator()( const T& a, const T& b) { return a == b; }
};
```
Function Objects

template <class T>
struct equals {
    bool operator()( const T& a, const T& b) { return a == b; }
};

Using a function object:
equals<int> int_equal;
if ( int_equal( 42, 42)) { ... }
Function Objects

template <class T>
struct equals {
    bool operator()( const T& a, const T& b) { return a == b; }
};

Using a function object:
equals<int> int_equal;
if (int_equal(42, 42)) { ... }

... or functor instantiation and functor call in one step:
if (equals<int>()(42, 42)) { ... }
Generic Contains Function Revisited

template <class InputIterator, class T, class Equal>
bool contains( InputIterator first, InputIterator beyond,
              const T& value, Equal eq = equals<T>()) {
    while ((first != beyond) && (! eq(*first, value)))
        ++first;
    return (first != beyond);
}
Generic Contains Function Revisited

template <class InputIterator, class T, class Equal>
bool contains( InputIterator first, InputIterator beyond,
               const T& value, Equal eq = equals<T>() ) {
  while ((first != beyond) && ( !eq(*first, value)))
    ++first;
  return (first != beyond);
}

int a[100];
// ... initialize elements of a.
bool found = contains( a, a+100, 42, equals<int>() );
Generic Contains Function Revisited

template <class InputIterator, class T, class Equal>
bool contains( InputIterator first, InputIterator beyond,
              const T& value, Equal eq = equals<T>() ) {
    while ((first != beyond) && ( ! eq(*first,value)))
        ++first;
    return (first != beyond);
}

Use a plain C function as parameter instead
bool equality( int a, int b ) { return a == b; } 

bool found = contains( a, a+100, 42, equality);
Benefits of Functors vs. Function Pointers

• Faster, because compile-time instantiation allows inlining etc.
• Can have state, e.g., see this epsilon neighborhood comparison:

```cpp
template <class T>
struct eps_equals {
  T epsilon;
  eps_equals( const T& eps ) : epsilon(eps) {}
  bool operator()( const T& a, const T& b ) {
    return (a-b <= epsilon) && (b-a <= epsilon);
  }
};
bool found = contains( a, a+100, 42, eps_equals<int>(1));
```
Adaptable Function Objects

• Adaptable function objects require in addition to regular function objects some local types that describe the result type and the argument types.

• A function pointer can be a valid model for a function object, but it cannot be a valid model of an adaptable function object.

```cpp
template <class T>
struct equals {
    typedef bool result_type;
    typedef T first_argument_type;
    typedef T second_argument_type;
    bool operator()( const T& a, const T& b) { return a == b; }
};
```
# Adaptable Function Objects

<table>
<thead>
<tr>
<th>Concept</th>
<th>Refinement of</th>
<th>Syntactic requirement model T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptable Generator</td>
<td>Generator</td>
<td>T::result_type</td>
</tr>
<tr>
<td>Adaptable Unary Function</td>
<td>Unary Function</td>
<td>T::result_type, T::argument_type</td>
</tr>
<tr>
<td>Adaptable Binary Function</td>
<td>Binary Function</td>
<td>T::result_type, T::first_argument_type, T::second_argument_type</td>
</tr>
<tr>
<td>Adaptable Predicate</td>
<td>Predicate,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adaptable Unary Function</td>
<td></td>
</tr>
<tr>
<td>Adaptable Binary Predicate</td>
<td>Binary Predicate,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adaptable Binary Function</td>
<td></td>
</tr>
</tbody>
</table>
Adaptable Function Objects

Simplify implementation with supporting base classes:
#include <functional>

template <class T>
struct equals : public std::binary_function<T,T,bool> {
    bool operator()( const T& a, const T& b) { return a == b; }
};

The definition of binary_function in the STL:
template <class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
Predicate Adaptor: unary_negate

template <class Predicate>
class unary_negate : public unary_function<
    typename Predicate::argument_type, bool> {

protected:
    Predicate pred;

public:
    explicit unary_negate( const Predicate& x) : pred(x) {}
    bool operator()(const typename Predicate::argument_type& x) const {
        return ! pred(x);
    }
};
Predicate Adaptor: unary_negate

A small helper function simplifies the use of the adaptor class:

template <class Predicate>
inline unary_negate< Predicate> not1( const Predicate& pred) {
    return unary_negate< Predicate>( pred);
}
Example: remove_if_divides

A short program in [Stepanov&Lee 95] makes us if this negator. The program copies all integers from cin to cout that cannot be divided by the integer parameter given to the program.

```cpp
int main( int argc, char** argv) {
    if ( argc != 2)
        throw( "usage: remove_if_divides integer\n");
    remove_copy_if( istream_iterator<int>(cin), istream_iterator<int>(),
                    ostream_iterator<int>(cout, "\n"),
                    not1( bind2nd( modulus<int>(), atoi( argv[1]))));
    return 0;
}
```

Instead of the helper function one would have to write the type:

```cpp
unary_negate<binder2nd<modulus<int> > > >(…)
```
Implementation of **bind2nd**

bind2nd is a helper function to create an binder2nd object

```cpp
template < class Operation, class Tp>
inline binder2nd< Operation>
bind2nd( const Operation& fn, const Tp& x) {
    typedef typename Operation::second_argument_type Arg2_type;
    return binder2nd< Operation>( fn, Arg2_type(x));
}
```
Implementation of `binder2nd<Op>`

`binder2nd` implements something similar to currying, i.e., it is a higher order function object.

```cpp
template <class Operation> struct binder2nd

  : public unary_function< typename Operation::first_argument_type,
                           typename Operation::result_type> {

  Operation op;
  typename Operation::second_argument_type value;

  binder2nd( const Operation& x,
             const typename Operation::second_argument_type& y)
              : op(x), value(y) {}

  typename Operation::result_type
  operator()(const typename Operation::first_argument_type& x) const {
    return op(x, value);
  }

};
```
Barton-Nackman Trick

We start with a simple class that, among other operations, provides an equality and an inequality comparison operator.

```cpp
struct A {
    bool operator == (const A& a) const;
    bool operator != (const A& a) const { return !(*this == a); }
};
```

Refactor generic inequality implementation into a base class.
Barton-Nackman Trick

Refactor generic inequality implementation into a base class. Problem is that base class needs to know the derived class. Let us provide it as template parameter.

```cpp
template <class T>
struct Inequality {
    bool operator != (const T& t) const {
        return ! (static_cast<const T&>(*this) == t);
    }
};

struct A : public Inequality<A> {
    bool operator == (const A& a) const;
};
```

There is a pitfall with name-lookup rules; don't use equal member function names in the base class and in the derived class!
Solving Mutual Dependencies between Class Templates

struct Edge;

struct Node {
    Edge * edge;
    // .... maybe more than one edge ....
};

struct Edge {
    Node * source;
    Node * dest;
};
Solving Mutual Dependencies between Class Templates

For a templated solution one would like to exchange each of the participating types independently and let the user assemble the final structure.

For example, a Node will be replaced by a Node’.
Solving Mutual Dependencies between Class Templates

template <class G>
struct Node {
    typedef typename G::Edge Edge;
    Edge* edge;
    // .... maybe some more edges ....
};

template <template <class G> class TNode, template <class G> class TEdge>
struct Graph {
    typedef Graph< TNode, TEdge> Self;
    typedef TNode<Self> Node;
    typedef TEdge<Self> Edge;
};

int main() {
    typedef Graph< Node, Edge> G;
    G::Node node;    G::Edge edge;
    node.edge = &edge;    edge.node = &node;
}
Solving Mutual Dependencies between Class Templates

template <class Graph>
struct Colored_node : public Node<Graph> {
    int color;
};

int main() {
    typedef Graph< Colored_node, Edge> G;
    G::Node node;
    G::Edge edge;
    node.edge = &edge;
    edge.node = &node;
    node.color = 3;
}