An Introduction to Generic Programming

Kuo-Hua Wang

Dept. of CSIE, Fu Jen Catholic University

2019/2 ~ 2019/6

Outline

- Lifting
- Concepts
- Models
- Specialization
- Conclusion
- Reference
  - http://www.generic-programming.org/
  - http://www.stroustrup.com/

What is Generic Programming?

- **Generic Programming** is a programming paradigm for developing efficient, reusable software libraries.
  - Generic Programming obtained its first major success when the Standard Template Library became part of the ANSI/ISO C++ standard.
- **Generic Programming** is a style of computer programming in which algorithms are written in terms of types *to-be-specified-later* that are then *instantiated* when needed for specific types provided as parameters.
- **Parametric Polymorphism**

Parametric Polymorphism

- In programming languages and type theory, **parametric polymorphism** is a way to make a language more expressive, while still maintaining full static type-safety. Using parametric polymorphism, a function or a data type can be written generically so that it can handle values *identically* without depending on their type.[1] Such functions and data types are called **generic functions** and **generic datatypes** respectively and form the basis of generic programming.

Templates (C++)

- **Templates** are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.
- Templates are of great utility to programmers in C++, especially when combined with multiple inheritance and operator overloading. The C++ Standard Library provides many useful functions within a framework of connected templates.

The Generic Programming Process

- The Generic Programming process focuses on **finding commonality (共性)** among similar implementations of the same algorithm.
- It provide abstractions in the form of **concepts (概念)** so that a single, generic algorithm can realize many concrete implementations.
- This process, called **lifting (提高)**, is repeated until the generic algorithm has reached a suitable level of abstraction, where it provides maximal reusability without sacrificing performance.

Lifting

- Lifting Basic Types
- Lifting Containers
- Lifting Iteration
- Review

Lifting Basic Types

```
int sum(int* array, int n)
{
   int result = 0;
   for (int i = 0; i < n; ++i)
      result = result + array[i];
   return result;
}
```

```
float sum(float* array, int n)
{
    float result = 0;
    for (int i = 0; i < n; ++i)
        result = result +
array[i];
    return result;
 }
```

Lifting Basic Types (Cont.)

```
template <typename T>
T sum(T* array, int n) {
    T result = 0;
    for (int i = 0; i < n; ++i)
        result = result + array[i];
    return result;
}
```

Using C++ Templates

Lifting Basic Types (Cont.)

```
template <typename T>
T sum(T* array, int n) {
    T result = 0;            // initialize it to 0
    for (int i = 0; i < n; ++i)
        result = result + array[i];  // operators + and =
    return result;     // copy constructor
}
```

Lifting Basic Types (Cont.)

```
template <typename T>
T sum(T* array, int n)
{
    T result = 0;
    for (int i = 0; i < n; ++i)
        result = result +
array[i];
    return result;
}
```

```
std::string
concatenate(std::string* array, int n)
{
    std::string result = "";
    for (int i = 0; i < n; ++i)
        result = result + array[i];
    return result;
}
```

Lifting Basic Types (Cont.)

```
// Requirements:
// T must have a default constructor that produces the identity value
// T must have an additive operator +
// T must have an assignment operator
// T must have a copy constructor
template<typename T>
T sum(T* array, int n) {
    T result = T();            // default constructor
    for (int i = 0; i < n; ++i)
        result = result + array[i];
    return result;   // copy constructor
}
```

Lifting Containers

```
// Requirements:
// T must have a default constructor that produces the identity value
// T must have an additive operator +
// T must have an assignment operator
// T must have a copy constructor
// Container must have an indexing operator [] that returns a T
template<typename Container, typename T>
T sum(const Container& array, int n) {
    T result = T();
    for (int i = 0; i < n; ++i)
        result = result + array[i];
    return result;
}
```

Lifting Iteration

```
template<typename T>
struct list_node {
    T value;
    list_node<T>* next;
};
template<typename T>
struct linked_list {
    list_node<T>* start;
};
template<typename T>
T sum(linked_list<T> list, int n) {
    T result = 0;
    for (list_node<T>* current = list.start; current != NULL; current = current->next)
        result = result + current->value;
    return result;
}
```

Lifting Iteration (Cont.)

```
Template <typename T>
T sum(T* array, int n) {
    T result = T();
    for (T* current = array; current != array + n; ++current)
        result = result + *current;
    return result;
}
```

Abstract the Requirements

// Requirements:

// T must have an additive operator +

// T must have an assignment operator

// T must have a copy constructor

// I must have an inequality operator !=

// I must have a copy constructor

// I must have an operation next() that moves to the next value in the sequence

// I must have an operation get() that returns the current value (of type T).

```
Template <typename I, typename T>
T sum(I start, I end, T init) {
    for (I current = start; current != end; current = next(current))
        init = init + get(current);
    return init;
}
```

Abstract the Requirements (Cont.)

```
template <typename T>
T*  next(T* p) { return ++p; }
template <typename T>
list_node<T>*  next(list_node<T>* n) { return n->next; }

template <typename T>
T get(T* p) { return *p; }
template <typename T>
T get(list_node<T>* n) { return n->value; }
```

Review

- The lifting process of Generic Programming integrates many concrete implementations of the same algorithm, teasing out the minimal requirements that algorithms place on their parameters.
- The requirements extracted during lifting can be combined and categorized into **concepts**, providing descriptions of the core abstractions in a problem domain.

Concepts

- Nested Requirements (巢狀需求)
- Associated Types (相關型別)
- Refinement (精煉、強化)

Concept (generic programming)

- In generic programming, a **concept** is a description of supported operations on a type, including syntax and semantics. In this way, concepts are related to abstract types but concepts do not require a subtype relationship.

Concepts (C++)

- **Concepts** are an extension to C++'s templates, published as an ISO Technical Specification ISO/IEC TS 19217:2015.[1] They are named boolean predicates on template parameters, evaluated at compile time. A concept may be associated with a template (class template, function template, or member function of a class template), in which case it serves as a *constraint*: it limits the set of arguments that are accepted as template parameters.

The Main Uses of Concepts

- Introducing type-checking to template programming
- Simplified compiler diagnostics for failed template instantiations
- Selecting function template overloads and class template specializations based on type properties
- Constraining automatic type deduction
- References
  - https://isocpp.org/
  - https://en.cppreference.com/w/cpp/experimental/constraints
  - Concepts: The Future of Generic Programming

What are Concepts?

- Concepts
  - bundle together coherent (連貫一致的) sets of requirements into a single entity.
  - describe a family of related abstractions (抽象概念) based on what those abstractions can do.
- Examples
  - an **Iterator** concept would describe abstractions that iterate over sequences of values (such as a pointer),
  - a **Socket** concept would describe abstractions that communicate data over a network (such as an IPv6 socket), and
  - a **Polygon** concept would describe abstractions that are closed plane figures (such triangles and octagons).

Introduction of Concepts

- Concepts are neither designed nor invented.
- Concepts are discovered through the process of lifting many algorithms within the same domain.
- The result of the lifting process is **a generic algorithm** and **a set of requirements**.

The result of lifting the sum algorithm.

```
// Requirements:
//  T must have an additive operator +
//  T must have an assignment operator
//  T must have a copy constructor
//  I must have an inequality operator !=
//  I must have a copy constructor
//  I must have an assignment operator
//  I must have an operation next() that moves to the next value in the sequence
//  I must have an operation get() that returns the current value (of type T).
template <typename I, typename T>
T sum(I start, I end, T init) {
    for (I current = start; current != end; current = next(current))
        init = init + get(current);        // + and = for type T, get() for type I
    return init;        // copy constructor for type T
}
```

The result of lifting an algorithm find that searches for a value in a sequence.

```
// Requirements:
//  T must have an equality operator ==
//  T must have a copy constructor
//  I must have an inequality operator !=
//  I must have a copy constructor
//  I must have an assignment operator
//  I must have an operation next() that moves to the next value in the sequence
//  I must have an operation get() that returns the current value (of type T).
template<typename I, typename T>
I find(I start, I end, T value)
{
    for (I current = start; current != end; current = next(current))
        if (get(current) == value)
            return current;
    return end;
}
```

ConceptName<T1, T2, …, TN>

Concept	Requirements
`CopyConstructible<T>`	`T` must have a copy constructor
`Assignable<T>`	`T` must have an assignment operator
`Addable<T>`	`T` must have an `operator+` that takes two `T` values and returns a `T`
`EqualityComparable<T>`	`T` must have an `operator==` comparing two `T`s and returning a `bool`. `T` must have an `operator!=` comparing two `T`s and returning a `bool`.
`Iterator<I, T>`	`I` must have an `operator==` comparing two `I`s and returning a `bool`. `I` must have an `operator!=` comparing two `I`s and returning a `bool`. `I` must have a copy constructor. `I` must have an assignment operator. `I` must have an operation `next()` that moves to the next value in the sequence. `I` must have an operation `get()` that returns the current value (of type `T`).

The specification of the requirements of sum() and find():

```
// Requirements: Addable<T>, Assignable<T>, CopyConstructible<T>, Iterator<I,T>
template <typename I, typename T>
T sum(I start, I end, T init);

// Requirements: EqualityComparable<T>, Assignable<T>, CopyConstructible<T>,
//               Iterator<I,T>
template <typename I, typename T>
I find(I start, I end, T value);
```

Nested Requirements

- Reuse those prior concepts in the definition of other concepts, by way of nested requirements.
- A nested requirement is when a concept references another concept as one of its own requirements.
  - Example:

    Iterator<I, T> concept requires EqualityComparable<I>

Concept	Requirements
`Iterator<I, T>`	`EqualityComparable<I>`, `CopyConstructible<I>`, `Assignable<I>` `I` must have an operation `next()` that moves to the next value in the sequence. `I` must have an operation `get()` that returns the current value (of type `T`).

Associated Types

The distance() function, which computes the length of a sequence.

```
// Requirements: Iterator<I,T>
template<typename I, typename T>
int distance(I start, I end) {
    int i = 0;
    for (; start != end; ++start) ++i;
    return i;
}
```

The Problem: there is no reference to T anywhere in the function signature.

Associated Types (Cont.)

- An updated Iterator concept:

Concept	Requirements
Iterator<I>	EqualityComparable<I>, CopyConstructible<I>, Assignable<I> I must have an operation next() that moves to the next value in the sequence. value_type is an associated type, accessible via iterator_traits<I>::value_type I must have an operation get() that returns the current value (of type value_type).

We can express the distance() algorithm more simply:

```
// Requirements: Iterator<I>
template<typename I>
int distance(I start, I end) {
    int i = 0;
    for (; start != end; ++start)  ++i;
    return i;
}
```

Associated Type:

iterator_traits<I>::value_type

Associated Types (Cont.)

- In C++, associated types are stored in class templates called **traits (特徵，特點，特性).**
- Traits are auxiliary class templates that can be specialized to retrieve the associated types for a particular use of the concept.

```
// Requirements: Iterator<I>, Addable<value_type>,
//                     Assignable<value_type>,  CopyConstructible<value_type>
template<typename I>
typename iterator_traits<I>::value_type
sum(I start, I end, typename iterator_traits<I>::value_type init)
{
    for (I current = start; current != end; current = next(current))
        init = init + get(current);
    return init;
}
```

Associated Types (Cont.)

- C++ *Specialization*
  - Example: the following class **template** ***partial specialization* (偏特化)** states that the value_type of a pointer T* (which is an iterator) is T:

```
template<typename T>
struct  iterator_traits<T*> {
      typedef  T value_type;
};
```

Refinement

- Nested Requirements
  - reuse concepts to describe other concepts.
- Concept refinement describes a hierarchical relationship between two concepts.
- If a concept C2 **refines** (精鍊、精製) a concept C1, then C2 includes all of the requirements of C1 and adds its own new requirements. So every C2 is also a C1, but C2 is more specific, and presumably enables more and better algorithms.

Concept	Requirements
`BidirectionalIterator<I>`	Refines `Iterator<I>` `I` must have an operation `prev()` that moves to the previous value in the sequence. `I` must have an operation `set()` that sets the current value.

```
// Requirements: BidirectionalIterator<BI>, Assignable<value_type>,
//                CopyConstructible<value_type>
template<typename BI>
void reverse(BI start, BI end) {
    while (start != end) {
        end = prev(end);
        if (start == end)  break;

        // Swap the values
        typename iterator_traits<BI>::value_type  tmp = get(start);
        set(start, get(end));
        set(end, tmp);

        start = next(start);
    }
}
```

Models

- Concepts describe a set of requirements, which are satisfied by a family of abstractions.
- The abstractions are typically data types or sets of data types, which we call **models**.
  - For instance, **a pointer** is a model of the **Iterator** concept; alternatively, we can say that a pointer models the Iterator concept.
- One of the most important aspects of Generic Programming is that the set of models for a given concept is neither known nor fixed.
- A given concept is written using a small, known set of models--say, nodes in a linked list and pointers into arrays--but will apply to many, many other data types, such as an pre-order iteration through a binary tree.

Specialization

- Concept refinement enables more and better algorithms, because the refining concept introduces more operations that describe richer abstractions.
  - Example: BidirectionalIterator allowed the efficient implementation of the reverse() algorithm.
- Iterator Concepts
  - Input Iterators
  - Output Iterators
  - Forward Iterators
  - Bidirectional Iterators
  - Random Access Iterator

Specialization (Cont.)

- Consider a Polygon concept.

```
// Requirements: Polygon<P>
template<typename P>
double circumference(P p) {
    double result = 0;
    for (int i = 0; i < num_sides(p); ++i)
        result += side_length(p, i);
    return result;
}
```

- A concept EqualateralPolygon that refines Polygon concept.

```
// Requirements: EquilateralPolygon<P>
template<typename P>
double circumference(P p) {
    return num_sides(p) * side_length(p, 0);
}
```

Concept-Based Overloading

In C++, this can be accomplished with a technique called **tag dispatching**.

Conclusion

- The Generic Programming Process
  - from the initial **lifting** of concrete implements into generic algorithms through **concept** analysis,
  - the mapping of diverse abstractions to concepts through **models**, and
  - finally the use of **specialization** to provide improved algorithms for more specific concepts.
- Generic Libraries written using the GP paradigm:
  - ConceptC++, an extension to C++ that provides drastically improved support for Generic Programming.
  - Generic Programming in C++, a guide to the various techniques and tricks used to implement generic libraries in C++.
  - The SGI Standard Template Library documentation, which provides documentation for all of the concepts, algorithms, and data structures in the STL.
- Concepts: The Future of Generic Programming

Generic Programming in C++: Concepts

Concepts

- Iterators, Containers, and Utility concepts from the Standard Template Library.
- Graph concepts for graph theory, from the Boost Graph Library.
- Matrix and Vector concepts for linear algebra, from the Matrix Template Library.
- Ring and Field concepts, from the Computation Geometry Algorithms Library.

Generic Programming in C++: Techniques

Outline

- Introduction
- Concepts
- Algorithms
- Traits
- Tag Dispatching
- Arbitrary Overloading
- Adaptors
- Concept Checking
- Archetypes

Introduction

- C++ can support Generic Programming very well through its template system, but to fully express the ideas of Generic Programming in C++ one must use a variety of template techniques.
  - Function Templates and Class Templates in C++

Concepts

- Concepts are documented as a set of requirements consisting of
  - Valid Expressions
    - C++ expressions which must be compiled successfully for the objects involved in the expression.
  - Associated Types
    - types that are related to the modeling type in that they participate in one or more of the valid expressions. **(trait class)**
  - Invariants
    - are run-time characteristics of the objects that must always be true (pre-conditions, post-condition)
  - Complexity Guarantees
    - Time complexity and Space complexity

Algorithms

- Generic algorithms in C++ are written using ***C++ templates***. Although C++ templates do not provide much type checking at the point of definition, they are type-safe at the point of instantiation and offer uncompromising performance.

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

Traits

- A **traits** class provides a way of associating information with a compile-time entity (a type, integral constant, or address).
- For example, the class template std::iterator_traits<T> looks something like this:

```
template <class Iterator>
struct iterator_traits {
    typedef … value_type;
    typedef … difference_type;
    typedef … pointer;
    typedef … reference;
};
```

Traits (Cont.)

- The traits' value_type gives generic code the type which the iterator is "pointing at“.
- The iterator_category can be used to select more efficient algorithms depending on the iterator's capabilities
- For an in-depth description of std::iterator_traits, see this page provided by cppreference.com.
- template< class T > class numeric_limits;

Tag Dispatching

- Tag dispatching is a way of using function overloading to effect concept-based overloading, and is often used hand-in-hand with traits classes.
- A good example of this synergy is the implementation of the std::advance() function in the C++ Standard Library, which increments an iterator n times.

```
namespace std {
    struct input_iterator_tag { };
    struct bidirectional_iterator_tag : input_iterator_tag { };
    struct random_access_iterator_tag : bidirectional_iterator_tag { };
    namespace detail {
        template <class InputIterator, class Distance>
        void advance_dispatch(InputIterator& i, Distance n, input_iterator_tag)
        {  while (n--) ++i;  }

        template <class BidirectionalIterator, class Distance>
        void advance_dispatch(BidirectionalIterator& i, Distance n,
                                        bidirectional_iterator_tag) {
            if (n >= 0)
                while (n--) ++i;
            else
                while (n++) --i;
        }
```

```
    template <class RandomAccessIterator, class Distance>
    void advance_dispatch(RandomAccessIterator& i, Distance n,
                                random_access_iterator_tag)
    { i += n; }
  }  // end of namespace detail
  template <class InputIterator, class Distance>
  void advance(InputIterator& i, Distance n) {
    typename iterator_traits<InputIterator>::iterator_category
                                                        category;
    detail::advance_dispatch(i, n, category);
  }
} // end of namespace std
```

Arbitrary Overloading

- Tag dispatching provides support for concept-based overloading of C++ templates, but is limited to decisions based on tags. However, one can use completely arbitrary template metaprograms to make overloading decisions using Substitution Failure Is Not An Error (SFINAE).

Arbitrary Overloading (Cont.)

```
template<bool, typename T = void>
struct enable_if {};
template<typename T>
struct enable_if<true, T> {
    typedef  T  type;
 };
```

- Essentially, enable_if<V, T>::type is T when V is true, but does not exist when V is false.

Arbitrary Overloading (Cont.)

- SFINAE means that when the compiler substitutes types into the declaration of a template, and that substitution fails for certain reasons (including not finding a member type), that template is silently eliminated from the set of function templates to be considered.
- For instance, say we want to write a function **sqrt** that works only for integral types. We could do so like this:

```
template<typename T>
typename enable_if<is_integral<T>::value, T>::type
sqrt(T x);
```

Arbitrary Overloading (Cont.)

- Since enable_if can be used with arbitrary template metafunctions, one can encode any kind of decision procedure to enable or disable certain templates, so long as the set of overloaded templates was coordinated so that only a single template is active for a given set of template arguments.
- For more information about enable_if and SFINAE, see the Boost enable_if library or read the following article.
- J. Järvi, J. Willcock, H. Hinnant, and A. Lumsdaine. Function Overloading Based on Arbitrary Properties of Types. C/C++ Users Journal, 21(6):25--32, June 2003.

Adaptors

- An ***adaptor*** is a class template which builds on another type or types to provide a new interface or behavioral variant.
- Adaptors are used when a type or set of types needs to model a concept whose interface is incompatible with the type(s).
- Examples:
  - std::reverse_iterator, which adapts an iterator type by reversing its motion upon increment/decrement, and
  - std::stack, which adapts a container to provide a simple stack interface.
- A more comprehensive review of the adaptors in the standard can be found here.

Concept Checking

- Concept checking is a way to detect errors in the use of templates early in their instantiation process, in the attempt of producing more readable error messages for users.
- To use concept checking, function and class templates are annotated with code that forces the instantiation of concept-checking classes.
- References
  - Boost Concept Check Library
  - C++ Concept Checking. Dr. Dobb's Journal, June 2001.
  - Concept checking: Binding parametric polymorphism in C++

Archetypes (原型)

- Archetypes provide improved type-checking for function and class templates in C++. While concept checking helps users by detecting when the template arguments do not model the concepts they should, archetypes help library designers by checking that the definition of a template only relies upon its types to provide behavior listed in its concept requirements

Archetypes (Cont.)

- ```
  template<typename InputIterator, typename T>
  InputIterator find(InputIterator first, InputIterator last, const T& value) {
      while (first < last && !(*first == value)) ++first;
      return first;
  }
  ```
- The error will not be detected until find() is instantiated with an iterator type that does not support the < operator.
  ```

Constraints and Concepts (since C++20)

Constraints and Concepts

- https://en.cppreference.com/w/cpp/language/constraints

A Tour of the STL

- A Simple Program
- Code using STL
- Copy for Input - Define an Iterator
- An Alternate Method
- Summary

A Simple Example

- Read text from the standard input,
- Breaking it into separate lines;
- Sort the Lines; and
- Write each line to the standard output.

Code using the STL

```
int main() {
    vector<string> V;      // Container
    string tmp;
    while ( getline(cin, tmp) )
        V.push_back(tmp);

    // sort its argument in ascending order
    sort( V.begin(), V.end() );     // Iterator & Algorithm
    copy( V.begin(), V.end(), ostream_iterator<string>(cout, “\n”) );
}
```

Copy for Input

- **Define an iterator that returns one line at a time.**

```
class line_iterator
{
    istream*  in;
    string   line;
    bool     at_end;
    void read( ) {
        if (*in)
           getline(*in, line);
        at_end = (*in) ? true : false;
    }
    ....    // next page
```

```
public:
    typedef  input_iterator_tag  iterator_category;
    typedef  string  value_type;
    typedef  ptrdiff_t  difference_type;
    typedef  const string*  pointer;
    typedef  const string&  reference;

    line_iterator( ) : in(&cin), at_end(false) { }
    line_iterator(istream& s) : in(&s) { read( ); }
    reference operator*( ) const { return line; }
    pointer operator->( ) const { return &line; }
    line_iterator operator++( ) {
         read( );
         return *this;
   }
   line_iterator operator++(int) {
         line_iterator  tmp = *this;
         read();
         return tmp;
   }
```

```
    bool operator==(const line_iterator& i) const {
        return (in == i.in && at_end == i.at_end) ||
                    (at_end == false && i.at_end == false);
    }

    bool operator!=(const line_iterator& i) const {
        return !(*this == i);
    }
};   // end  of class line_iterator

int  main()
{
    line_iterator  iter(cin);
    line_iterator  end_of_file;
    vector<string>  V(iter, end_of_file);

    sort( V.begin(), V.end() ); // sort( V.begin(), V.end(),
greater<string>() );
    copy( V.begin(), V.end(), ostream_iterator<string>(cout, “\n”));
}
```

An Alternate Method

- Two Tables
  - One for the string table (STbl)
  - One for pointers into the string table (STbl)
- Each line is represented as **a** **pair** **of iterators** into the table.
  - One pointing to the first character of the line
  - One pointing just beyond the last character.
- Tell `sort()` how to compare two such elements – using the < operator to compare them no longer works.
  - Use user-defined `strtab_cmp()` function object.

```
struct  strtab_cmp
{
    typedef  vector<char>::iterator  strtab_iterator;
    bool  operator()( const pair<strtab_iterator, strtab_iterator>& x,
                      const pair<strtab_iterator, strtab_iterator>& y ) const {
        return lexicographical_compare( x.first, x.second, y.first, y.second );
    }
};

struct  strtab_print
{
    ostream&  out;
    strtab_print(ostream&  os) : out (os) { }
    typedef  vector<char>::iterator  strtab_iterator;

    void  operator()( const pair<strtab_iterator, strtab_iterator>&  s ) const {
         copy( s.first, s.second, ostream_iterator<char>(cout) );
    }
}
```

Summary

- Standard Template Library (STL)
  - Generic Algorithms – sort, find, and lexicographical_compare
  - Iterators – istream_iterator and ostream_iterator
  - Containers – vector
  - Function Objects – less and greater
- Important Aspects of Using STL
  - *To use the STL is to extend it.*
  - *The STL algorithms are decoupled from the containers.*
  - *The STL is extensible and customizable without inheritance.*
  - *Abstraction need not mean inefficiency.*

Algorithms and Ranges

Introduction

- Iterators are the most important innovation in the STL, and they are what makes it possible to decouple algorithms from the data structures (containers) they operate on.
- *Concepts*, *Modeling*, and *Refinement*

Linear Search (or Sequential Search)

- Knuth's terminology [Knu98b]
  - Finding a particular value in a linear collection of elements.

- Linear Search in C

```
char*  strchr(char* s, int c)
{
    while (*s != '\0' && *s != c)
        ++s;
    return *s == c ? s : (char*) 0;
}
```

Linear Search (Cont.)

- Answer the following questions.
  - How do we specify which sequence we're searching through?
  - How do we represent a positon within the sequence?
  - How do we advance to the next element?
  - How do we know when we have reached the end of the sequence?
  - What value do we return as an indication of failure?

Linear Search (Cont.)

- ```
  char*  find1(char  *first, char  *last, int  c)
  {
      while (first != last && *first !=c)
          ++first;
      return first;
  }
  ```
- ```
  char  A[N];
  ...
  char* result = find1(A, A+N, c);
  if (result == A + N) { // The search failed.
  } else {
      // The search succeeded
  }
  ```

Linear Search (Cont.)

- C has three different kinds of pointers
  - Ordinary valid pointers, like &A[0], which you can dereference
  - Invalid Pointers, like NULL (*singular* pointers); and
  - Past-the-end pointers that you can't dereference but you can use in pointer arithmetic.

Ranges

- ***Range***[first, last)
  - Consists of the pointers from first up to but not including last
  - It refers to the pointers first, first+1,..., last-1 ***(range of pointers)***
  - It refers to the elements *first, *(first+1),.... *(last-1) ***(range of elements)***
  - It is ***Half-Open Intervals***
- **Empty range [A, A)**

Linear Search in C++

- ```
  template <class T>
  T*  find2(T*  first, T*  last, T value);
  ```
- The STL uses

  ```
  template <class Iterator, class T>
  Iterator find(Iterator first, Iterator last, const T& value)
  {
      while (first != last && *first != value)
          ++first;
      return first;
  }
  ```
  ```

Search Through a Linked List

- ```
  struct int_node {
      int val;
      int_node* next;
  }
  ```
- The code for traversal looks something like the following:
  ```
  int node* p;
  for (p = list_head; p != NULL; p = p->next)
      // Do something.
  ```
- Use ***node wrapper*** for int_node.
  ```

Node Wrapper for int_node

```
template <class Node>
struct node_wrap {
    Node* ptr;

    node_wrap(Node* p = nullptr) : ptr(p) { }

    Node& operator*() const { return *ptr; }
    Node* operator->() const { return ptr; }

    node_wrap& operator++() { ptr = ptr->next; return *this; }
    node_wrap operator++(int) { node_wrap tmp = *this; ptr = ptr->next; return tmp; }

    bool operator==(const node_wrap& i) const { return ptr == i.ptr; }
    bool operator!=(const node_wrap& i) const { return ptr != i.ptr; }
}
```

Node Wrapper (Cont.)

- Define an equality operator == which can compare an int_node to an int.

```
template <class NODE>
bool operator ==(const NODE& node, int n) {
    return node.value == n;
}
template <class NODE>
bool operator !=(const NODE& node, int n) {
    return !(node == n);
}
```

- We can reuse

```
find(node_wrap<int_node>(list_head), node_wrap<int_node>(),
val)
```

Concepts and Modeling

- template <class Iterator, class T>
  Iterator find(Iterator first, Iterator last, const T& value)
- Iterators are a fundamental part of the STL.
- Iterator must be an **Input Iterator**.
- Input iterator is a ***Concept***.
- A concept describes **a set of requirements on a type**.
- When a specific type satisfies all of those requirements, we say that it is a ***model*** of that concept.
- For example, char* and node_wrap are models of input iterator.

What is a Concepts?

- First, a concept can be thought of as **a list of type requirements**.
- Second, a concept can be thought of as **a set of types**.
- Third, a concept can be thought of as **a list of valid programs**.

Concept Requirements

- The requirements are ***a set of valid expressions***. For example, if Iterator is a model of Input Iterator and i is an object of type Iterator, then *i is a valid expression.

Basic Concepts

- Assignable
  - X x(y);   x = 8;   or   tmp = y, x = tmp;
- Default Constructable
  - T();   T t;
- Equality Comparable
  - x == y   or   x != y
- LessThan Comparable
  - x < y   or   x > y
- **Regular Type**
  - is one that is a model of Assignable, Default Constuctable, Equality Comparable and one in which these expressions interact in the expected way.
  - Most of the C++ types are regular types, and so are almost all of the types defined in the STL.

Iterators

- Input Iterators
  - **read only**, single pass algorithms, operator++, operator*
- Output Iterators
  - **write only**, single pass algorithms, operator++, operator*
- Forward Iterators
  - It is refinement of Input and Output Iterators.
  - It allowed two iterators in the range of iterators.
- Bidirectional Iterators
  - operator++, operator--, multi-pass algorithms
- Random Access Iterators
  - It includes all operations of pointer arithmetic, ++, --, p[n], p+n, p-n, (p1-p2), (p1 < p2)

Input Iterators

- find()
  template <class InputIterator, class EqualityComparable>
  InputIterator find(InputIterator first, InputIterator last,
  const EqualityComparable& value);
- Input Iterators are similar to some pointers
  - *dereferenceable*(*), *past the end*, *singular* (null)
  - Compare the equality of two input iterators.
  - Input Iterator can be copied (copy constructor) and assignable.
  - Each input iterator has one **associated value type** (the type of objects it points to)
  - ++p and p++
- Linear search is a **single pass** algorithm.
- The other algorithms using input iterators are
  - find_if, equal, partial_sum, random_sample, set_intersection

Output Iterators

- copy()

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                                      OutputIterator result)
{
     for ( ; first != last; ++result, ++first)
          *result = *first;
     return result;
}
```

Output Iterators (Cont.)

- Output Iterators
  - It can be copied and assignable.
  - **Write-Only**, *p = x;
  - ++p, p++
  - It supports **single-pass** algorithms.
  - There are no two output iterators pointing to the same range.
- The other algorithms using output iterators are
  - copy, transform, merge.
- The iterator classes using output iterators are
  - insert_iterator, front_insert_iterator, back_insert_iterator
  - ostream_iterator

Output Iterators (Cont.)

- ```
  template <classs T> class ostream_iterator {
  private:
      ostream*  os;
      const char* string;
  public:
      ostream_iterator(ostream& s, const char* c = 0) : os(&s), string(c) { }
      ostream_iterator(const ostream_iterator& i)
            : os(i.os),  string(i.string)  { }
      ostream_iterator&  operator=(const ostream_iterator  i) {
            os = i.os;
            string = i.string;
            return *this;
      }
      ostream_iterator<T>& operator=(const T& value) { // *p = x performs formatted
            *os << value;                            // output of x onto an ostream;
            if (string)  *os << string;
            return *this;
      }
      ostream_iterator<T>& operator*()  { return *this; }
      ostream_iterator<T>& operator++()  { return *this; }
      ostream_iterator<T>& operator++(int)  { return *this; }
  };
  ```
  ```

Forward Iterators

- Forward iterator is the refinement of input iterator and output iterator.
- It supports multi-pass algorithms.
- It supports reading and writing in the same range.

```
template <class ForwaredIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
                const T& old_value, const T& new_value)
{
    for( ;  first != last; ++first)
        if (*first == old_value)
            *first = new_value;
}
```

Forward Iterators (Cont.)

- Search that two adjacent elements have the same value.

```
template <class ForwaredIterator>
ForwareIterator
adjacent_find(ForwardIterator first, ForwardIterator last)
{
    if (first == last)
        return last;
    ForwardIterator  next = first;
    while (++next != last) {
        if (*first == *next)
            return first;
        first = next;
    }
    return last;
}
```

Bidirectional Iterators

- It supports multi-pass algorithms.
- reverse_copy()

```
template <class BidirectionalIteator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator  first,
                            BidirectionalIterator  last,
                            OutputIterator  result) {
    while (first != last) {
        --last;
        *result = *last;
        ++result;
    }
    return result;
}
```

Random Access Iterators

- It supports all operations of C pointers.
  ++p, p++, --p, p--, p+n, p-n, p[n], (p1-p2) and (p1 < p2)
- It can access any element in constant time. (O(1))
- sort()

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

-

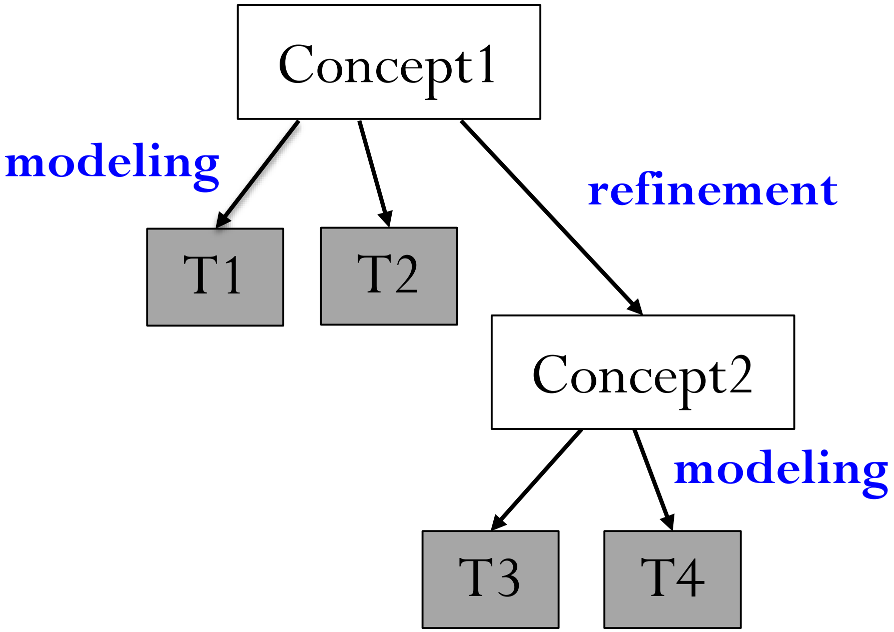
```
template <class RandomAccessIterator, class StrictWeakOrdering>
void sort(RandomAccessIterator first, RandomAccessIterator last,
          StrictWeakOrdering comp);
```

Refinement (精煉、強化)

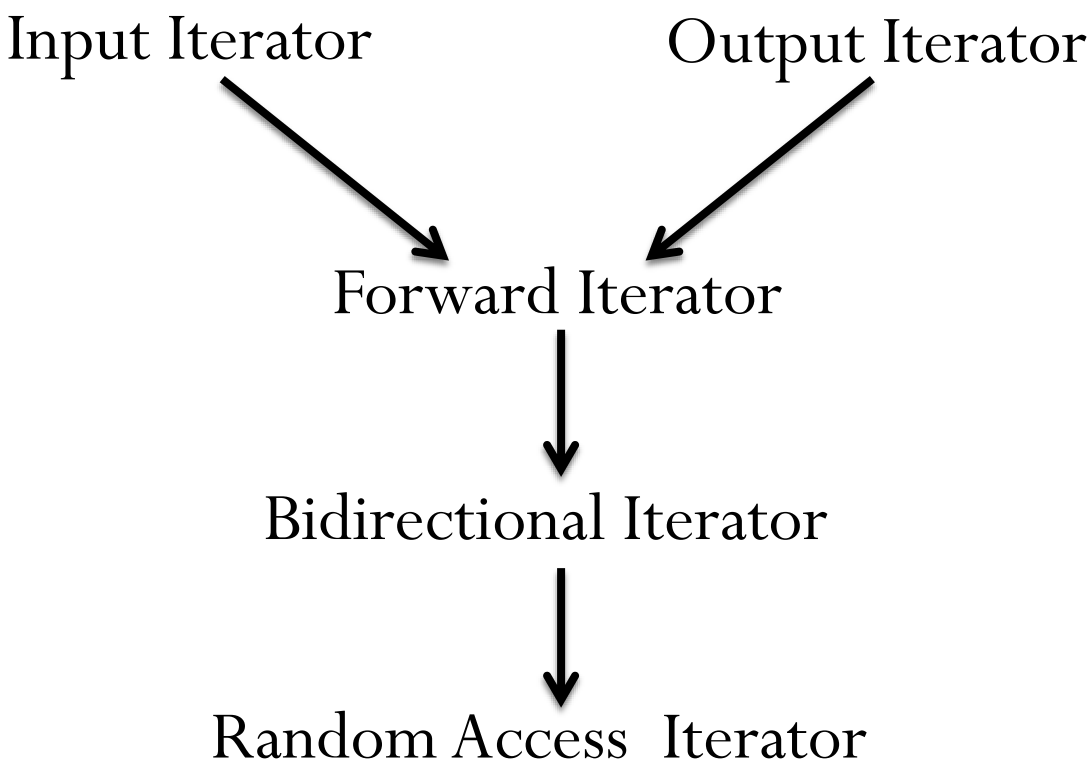
- A concept C2 is a **refinement** of the concept C1 if C2 provides all of the functionality of C1 and possibly additional functionality as well.
- Modeling and refinement satisfy three crucial properties:
  1. **Reflexivity**. Every concept C is a refinement of itself.
  2. **Containment**. If a type X is a model of the concept C2 and if C2 is a refinement of the concept C1, the X is also a model of C1.
  3. **Transitivity**. If C3 is a refinement of C2 and C2 is a refinement of C1, then C3 is a refinement of C1

Refinement (Cont.)

- Modeling and Refinements



Summary


Input Iterator
Output Iterator
Forward Iterator
Bidirectional Iterator
Random Access Iterator

More About Iterators

Iterator Traits and Associated Types

- **Value Types (數值型別)**
  - **Problem:** Given a function f(I), where I is an iterator.
    How to declare a tempt variable with 「I's value type」?
  - **Solution:** Using C++ type inference (型別推斷).

```
template  <class I, class T>
void  f_impl(I iter, T t)
{
     T tmp;     //  T is I's value type
     ...
}
template  <class I>  inline void f(I  iter)   // forwarding  function
{
     f_impl(iter,  *iter);
}
```

  - **Another Issue:**
    How to declare a return value's type with 「I's value type」?

Value Types (Cont.)

- The Second Solution:

  Using C++'s nested type (巢狀型別）declaration. In the iterator class, declare it's value type.

```
template <class Node> struct node_wrap {
    typedef  Node  value_type;
    Node*  ptr;
    ...
}
```

- We can declare I's value type using typename I::value_type.
- It still has a problem for normal pointers like int *.

Value Types (Cont.)

- Solution:
  Define an auxiliary class, iterator_traits:

```
template <class Iterator>
struct iterator_traits {
   typedef  typename  Iterator::value_type  value_type;
}
```

- Using `typename iterator_traits<I>::value_type` to get the iterator I's value type.
- Template Specialization (for each pointer type)

```
template <class T>
struct iterator_traits<T*> {   // it can be <const T*>, too
    typedef T value_type;
}
```

- Three versions of iterator_traits: (1) T   (2) T* (3) const T*

Value Types – An Example

- ```
  template  <class InputIterator>
  typename  iterator_traits<InputIterator>::value_type
  sum_nonempty(InputIterator first, InputIterator last)
  {
      typename iterator_traits<InputIterator>::value_type
      result = *first++;
      for ( ; first != last; ++first)
          result += *first;
      return result;
  }
  ```
  ```

Difference Types (差距型別)

- I: a model of Random Access Iterator
  p1, p2: the values of type I
  p2–p1 is the distance between p1 and p2.
- What type of the execution result p2-p1?
  Answer: It is the type ptrdiff_t which has been defined by C/C++. (ptrdiff_t is 32-bit unsigned number)
- It is not good enough to traverse big data files (database) which their sizes are over terabytes.
  Why? Think about the range of 32-bit unsigned numbers.

Difference Types (Cont.)

- Define difference_type as a nested type in iterator class.
- Example:

```
template  <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count(InputIterator  first, InputIterator  last, const T& x)
{
    typename iterator_traits<InputIterator>::difference_type n = 0;
    for ( ; first != last; ++first)
        if (*first == x)
            ++n;
    return n;
}
```

Reference Type and Pointer Type

- p: a Forward Iterator which points to an Object of type T
  How about *p?
  Answer:
  *p is not just returning an object of type T, it must return a **lvalue(左值)** which can be placed in the left side for an assignment operator (=). *p = …;
- C++ functions may return a reference as a lvalue.
- Example: the node_wrap calss
  the operator* of node_wrap returns Node& (reference), the operator-> returns Node* (pointer).

Dispatching Algorithm and Iterator Tags

- *Concept Overloading*
  - It often turns out that an algorithm has a sensible definition for one iterator concept, but there is a different way to define it for a refinement of that concept.
  - For the same algorithm, the time complexity depends on the used iterator concepts.

Example: advance()

- ```
  template  <class InputIterator, class Distance>
  void advance_II(InputIterator& i, Distance n)
  {
      for ( ; n > 0; --n, ++i) { }
  }
  ```
- ```
  template  <class BidirectionalIterator, class Distance)
  void advance_BI(BidirecctionalIterator& i, Distance n)
  {
      if (n >= 0)
          for ( ; n > 0; --n, ++i) { }
      else
           for ( ; n < 0; ++n, --i) { }
  }
  ```
- ```
  template  <class RandomIterator, class Distance)
  void advance_RAI(RandomIterator& I, Distance n)
  {
      I += n;
  }
  ```
  ```

Cont.

- template <class InputIterator, class Distance>

```
void advance(InputIterator& I, Distance n)
{
    if (is_random_access_iterator(i))
        advance_RAI(i, n);
    else if (is_bidirectinal_iterator(i))
        advance_BI(i, n);
    else
        advance_II(i, n);
}
```

```
template  <class InputIterator, class Distance>
void advance (InputIterator& i, Distance n, input_iterator_tag)
{
      // Same implementation as advance_II
}
template  <class ForwardIterator, class Distance)
void advance(ForwardIterator& i, Distance n, forward_iterator_tag)
{
     advance(i, n, input_iterator_tag);
}
template  <class BidirectionalIterator, class Distance)
void advance(BidirecctionalIterator& i, Distance n, bidirectional_iterator_tag)
{
     // Same implementation as advance_BI
}


template  <class RandomIterator, class Distance)
void advance(RandomIterator& I, Distance n, random_access_iterator_tag)
{
     // Same implementation as advance_RAI
}
```

- ```
  template  <class InputIterator, class Distance>
  inline void advance(InputIterator& i, Distance n)
  {
        advance(i, n, typename iterator_trait<i>::iterator_category());
  }
  ```

- The STL defines the five tag types as empty classes.

```
struct input_iterator_tag { };
struct output_iterator_tag { };
struct forward_iterator_tag : public input_iterator_tag { };
struct bidirectional_iterator_tag : public forward_iterator_tag { };
struct random_access_iterator_tag :
          public bidirectional_iterator_tag { };
```
```

Putting It All Together

```
template <class Iterator>
struct iterator_traits {
   typedef typename Iterator::iterator_category  iterator_category;
   typedef typename Iterator::value_type  value_tye;
   typedef typename Iterator::difference_type  difference_type;
   typedef typename Iterator::pointer  pointer;
   typedef typename Iterator::reference  reference;
};


template <class T>
struct iterator_traits<T*>{
   typedef random_access_iterator_tag  iterator_category;
   typedef T  value_tye;
   typedef ptrdiff_t  difference_type;
   typedef T*  pointer;
   typedef T&  reference;
};
```

Putting It All Together (Cont.)

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

Putting It All Together (Cont.)

```
template <class Category,
          class Value,
          class Distance = ptrdiff_t,
          class Pointer = T*,
          class Reference = T&>
struct iterator
{
   typedef Category    iterator_category;
   typedef Value       value_type;
   typedef Distance    difference_type;
   typedef Pointer     pointer;
   typedef Reference   reference;
};
```

Defining New Components

- template <class Node, class Reference, class Pointer>

```
template <class Node, class Reference, class Pointer>
struct node_wrap_base
    : public iterator<forward_iterator_tag, Node,
                      ptrdiff_t, Pointer, Reference>
{
    typedef node_wrap_base<Node, Node&, Node*>  iterator;
    typedef node_wrap_base<Node, const Node&,
                           const Node*>  const_iterator;
    Node *ptr;

    node_wrap_base(Pointer p = 0) : ptr(p) { }
    node_wrap_base(iterator x) : ptr(x.ptr) { }

    Reference operator*() const { return *ptr; }
    Pointer operator->() const { return ptr; }

    void incr()  {  ptr = ptr->next;  }

    bool operator==(const node_wrap_base& x) const { return ptr == x.ptr; }
    bool operator!=(const node_wrap_base& x) const { return ptr != x.ptr; }
}
```

Cont.

- ```
  template <class Node>
  struct node_wrap : public node_wrap_base<Node, Node&, Node*>
  {
      node_wrap(Node* p = 0) : node_wrap_base(p) { }
      node_wrap(const node_wrap<Node>& x) : node_wrap_base(x) { }

      node_wrap& opeator++()
        {    incr(); return this;  }
      node_wrap operator++(int)
        {   node_wrap tmp = *this;   incr();  return tmp; }
  };
  ```
- ```
  template <class Node>
  struct const_node_wrap
       : public node_wrap_base<Node, const Node&, const Node*>
  {
      const_node_wrap(const Node* p = 0) : node_wrap_base(p) { }
      const_node_wrap(const node_wrap<Node>& x) : node_wrap_base(x) { }

      const_node_wrap& opeator++()
        {    incr(); return this;  }
      const_node_wrap operator++(int)
        {   node_wrap tmp = *this;   incr();  return tmp; }
  };
  ```

Iterator Adaptors

- Broadly speaking, an adaptor is anything that transforms one interface into another.
- The node_wrap class is in a sense an adaptor, in that it provides an STL interface for a linked list that may have been written without any thought of iterators or the STL.
- STL iterator adaptors
  - front_insert_iterator
  - back_insert_iterator
  - reverse_iterator

Advice for Defining an Iterator

- Use the iterator requirements as a checklist.
- Be careful about whether the iterator is supposed to be constant or mutable.
  - T&, T* vs. const T&, const T*.
- You must make sure that iterator_traits is defined appropriately.
- You should provide as many iterator operators as you can without loss of efficiency.
- For forward iterators in particular, you should make sure to obey the fundamtal axiom of iterator equality: Two iterators are equal if and only if they point to the same object.

Advice for Defining an Algorithm

- Assume as little as possible. It means that your algorithm can be used with many difference kinds of iterator as possible.
- Use the dispatching technique to improve the efficiency of your algorithm.
- Consult the iterator requirements for the template parameters for the algorithm.
- Test your algorithm using a concrete type that adheres as closely as possible to that concept.

Summary

- The central feature of the STL is generic algorithms on ranges (range of iterators).
- The STL's five iterator concepts are the glue that binds an algorithm to the data structure it operates on.
- Reuse existing algorithms with new data structures.
- Reuse existing data structures with new algorithms.
- Iterators, along with functions objects allow even simple algorithms to be reused in very diverse context.

Function Objects

Generalization of Linear Search

- In Chapter 6 of *The Art of Computer Programming*,
  - In general, we shall suppose that a set of *N* records has been stored, and the problem is to locate the appropriate one. As in the case of sorting, we assume that each record includes a special field called it's *key*,… Algorithms for searching are presented with a so-called *argument*, *K*, and the problem is to find which record has *K* as its key.
- template <class InputIterator, class Predicate>

```
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred)
{
    while (first != last && !pred(*first))
        ++first;
    return first;
}
```

Template parameter Predicate:

It is a ***function object***, or ***functor***.

Cont.

- The simplest kind of function object is an ordinary function pointer.
  - bool is_even(int x) { return (x & 1) == 0; }
  - find_if(f, l, is_even);
  - A pointer to is_even has the type bool (*)(int).
- Use function call operator, operator().

```
template <class Number> struct even
{
    bool operator()(Number x) const { return (x & 1) ==
0; }
};
find_if(f, l, even<int>());
```

Cont.

- Function objects are not restricted to this simple form. A function object can be a fully general class, and like any other class, it can have *member functions* and *member variables*.

```
struct last_name_is
{
    string value;       // local state
    last_name_is(const string& val) : value(val) { }
    bool operator(const Person& x) const {
        return x.last_name == value;
    }
};
find_if(first, last, last_name_is("Smith"));
```

Function Object Concepts

- The fundamental requirement of any function object concept is that, if f is a function object, it is possible to apply operator() to f.

Unary and Binary Function Objects

- template <class ForwardIterator, class BinaryPredicate>

```
ForwardIterator
adjacent_find(ForwardIterator first, ForwardIterator last,
                   BinaryPredicate pred)
{
    if (first == last)
        return last;
    ForwardIterator  next = first;
    while (++nest != last) {
        if (pred(*first, *next))
            return first;
        first = next;
    }
    return last;
}
```

Predicates and Binary Predicates

- template <class InputIterator, class OutputIterator,
                    class UnaryFunction>
  OutputIterator transform(InputIterator first, InputIterator last,
                                                  OutputIterator result, UnaryFunction
  f)
  {
      while (first != last) *result++ = f(*first++);
      return result;
  }

Associated Types

- ```
  template <class Arg, class Result>
  struct unary_function {
      typedef Arg      argument_type;
      typedef Result   result_type;
  };

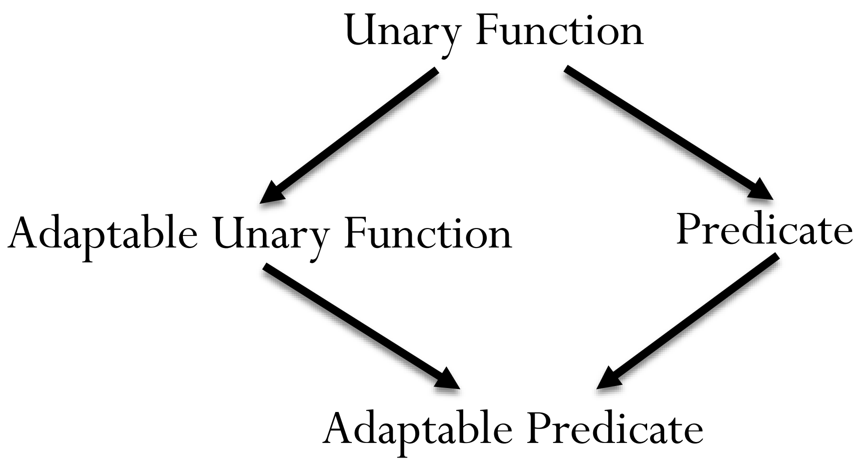
  template <class Arg1, class Arg2, class Result>
  struct binary_function {
      typedef Arg1     first_argument_type;
      typedef Arg2     second_argument_type;
      typedef Result   result_type;
  };
  ```
  ```

Associated Types (Cont.)

- ```
  template <class Number>
  struct even : public unary_function<Number, bool>
  {
      bool operator()(Number x) const { return (x & 1) == 0; }
  };
  ```
  ```

Cont.

- Concepts of Function Objects with Single Argument



Function Object Adaptors

- An *adaptor* is any component that transforms one interface into another.
  - Example: `reverse_iterator` is an iterator adaptor.
- Function object adaptors are particular tempting.
  - **Function composition** is a basic operation of mathematics and computer science.
  - Defining lots of *ad hoc* function object classes is a nuisance. A relatively small number of well-chosen function adaptors makes it possible to build complicated operations out of simpler ones.
  - Many useful function object adaptors are very easy to define.

Function Object Adaptors (Cont.)

```
template <class AdaptablePredicate>
class unary_negatge
{
  private:
      AdaptablePredicate pred;
  public:
      typedef typename AdaptablePredicate::argument_type
                 argument_type;
      typedef typename AdaptablePredicate::result_type
                 result_type;

      unary_negate(const AdaptablePredicate& x) : pred(x) { }
      bool operator() (const argument_type& x) const {
           return !pred(x);
      }
 };
```

Cont.

- Define a tiny helper function that makes it easier to create unary_negatge objects:

```
template <class AdaptablePredicate>
inline unary_negate<AdaptablePrediate>
not1(const AdaptablePredicate& pred)
{
    return unary_negate<Predicate>(pred);
}
```

- `find_if(first, last, not1(even<int>()));`

pointer_to_unary_function

- It transforms an ordinary function pointer to an Adaptable Unary Function.
- ```
  template <class Arg, class Result>
  class pointer_to_unary_function : public unary_function<Arg, Result> {
  private:
      Result (*ptr)(Arg);
  public:
      pointer_to_unary_function()  { }
      pointer_to_unary_function(Result (*x)(Arg)) : ptr(x) { }
      Result opertor()(Arg x) const { return ptr(x); {
  };
  template <class Arg, class Result>
  inline pointer_to_unary_function<Arg, Result>
  ptr_fun(Result (*x)(Arg)) {
       return pointer_to_unary_function<Arg, Result>(x);
  }
  ```
  ```

Predefined Function Objects

- The basic arithmetic operations defined in the STL:
  - Adaptable Binary Function
    plus, minus, multiplies, divides, modules , and
  - Adaptabler Unary Function
    negate.
- The basic comparison operations:
  - equal_to, not_equal_to, greater, less, greater_equal, and less_equal.
- The concept Strict_Weak_Ordering is a refinement of BinaryPredicate.

Summary

- The reason that function objects are useful is that they allow generic algorithms to be even more general. They make it possible to parameterize policy.
- Function objects, like most of the STL, are useful as an adjunct (附屬品) to algorithms on liner range.

Containers
容器

A Simple Container

- Array is the simplest data structure that can contain a range. The iterators with arrays are pointers.
  - Arrays naturally adhere to the idea of a range. If A is an array with N elements. All elements are contained in the range [A, A+N).
  - Array can be allocated on the stack. A declaration like
    `int A[10];` doesn't involve any dynamic allocation.
  - Arrays are efficient because it doesn't require multiple levels of indirection to access an array element.
  - Arrays have fixed size that is known at compile time.
  - Arrays have a convenient initialization syntax like below
    `int A[6] = {1, 4, 2, 8, 5, 7};`
- Think about the disadvantages of Arrays.
  - fixed size, can not find that past-the-end iterator directly (A+N)
  - no copy constructors or assignment operators
  - It is impossible to pass an array to a function by value. Array types usually "decay" to pointer types.

An Array Class

- ```
  template <class T, size_t N>
  struct block {
      typedef T value_type;
      typedef value_type* pointer;
      typedef const valule_type* const_pointer;
      typedef value_type& reference;
      typedef const value_type& const_reference;
      typedef ptrdiff_t differnce_type;      // signed
      typedef size_t size_type;              // unsigned
      typedef pointer iterator;
      typedef cont_pointer cont_iterator;

      iterator begin() { return data; }
      iterator end() { return data + N; }
      const_iterator begin() const { return data; }
      const_iterator end() const  { return data + N; }
      reference operator [] (size_type n) { return data[n]; }
      const_reference operator []  const (size_type n) { return data[n]; }

      size_type size() const { return N; }

      T data[N];
  };
  ```
  ```

An Array Class (Cont.)

- You declare a block of ten integers as block<int, 10> A;
- All of the block's elements are contained in the range [A.begin(), A.end());
- A container like block has associated types, just as iterators do.
- In C, you can iterate through an array using T* or const T*. We define two nested types, iterator and const_iterator.
- Two versions of the operator [] member functions.
- All of block's nested types:
  - iterator and const_iterator
  - value_type, differnce_type, size_type
  - pointer and const_pointer
  - reference and const_reference

How It Works

- `block<int, 10>`
  - the number of elements in a block is part of its type.
  - It is a completely different type from a `block<int, 12>`
  - `10` is called *nontype template parameters*. The template parameter `N` is a compile-time constant.
- It has no constructors, no destructor, no assignment operator.
- All of its members, including the array data itself , are public
- `block<int, 6> A = {1, 4, 2, 8, 5, 7};`

Finishing Touches

- *Regular Type*
- Block is a model of two o f those concepts: Assignable and Default Constructible. It is not, however, a model of Equality Comparable or LessThan Comparable.
  - We can write x == y or x < y, if x and y are two objects of type block<T, N>.

Finishing Touches (Cont.)

- ```
  template <class T, size_t N>
  bool operator ==(const block<T, N>& x, const block<T, N>& y)
  {
      for (size_t i = 0; i < N; ++i)
          if (x.data[i] != y.data[i])
              return false;
      return true;
  }
  ```

- ```
  template <class T, size_t N>
  bool operator <(const block<T, N>& x, const block<T, N>& y)
  {
      for (size_t i = 0; i < N; ++i)
          if (x.data[i] < y.data[i])
              return true;
          else if (x.data[i] > y.data[i])
              return false;
      return false;
  }
  ```

Finishing Touches (Cont.)

- ```
  template <class T, size_t N>
  struct block
  {
      ...
      typedef reverse_iterator<const_iterator> const_reverse_iterator;
      typedef reverse_iterator<iterator> reverse_iterator;

      reverse_iterator rbegin() { return reverse_iterator(end()); }
      reverse_iterator rend() { return reverse_iterator(begin()); }

       const_reverse_iterator rbegin() const {
            return const_reverse_iterator(end());
      }
      const_reverse_iterator rend() {
            return reverse_iterator(begin());
      }
  };
  ```
  ```

Finishing Touches (Cont.)

- ```
  template <class T, size_t N>
  struct block
  {
      ...
      size_type max_size() const { return N; }
      bool empty() const { return N == 0; }

      void swap(block& x) {
          for (size_t i = 0; i < N; ++i)
              std::swap(data[i], x.data[i]);
      }
  };
  ```
  ```

Container Concepts

- A block has three main areas of functionality.
  - It contains elements.
  - It provides access to those elements.
  - It supplies the operations that are necessary for a block to be a regular type.

Containment of Elements

- Two containers can't overlap, and an element can't be belonged to more than one container. You can put two different copies in two different containers, of course; the restriction is one objects, not values. Ownership, however, can't be shared.
- An element's lifetime can't extend beyond the lifetime of the container that it is a part of. An element is created no earlier than the when the container is constructed, and it is destroyed no later than when the container is destroyed.
- A container can be a *fixed size* container like block, or it can be a variable size container where you can create and/or destroy elements after the container is created. Even a variable size container "owns" its elements, and they are destroyed by the container's destructor.

Iterators

- 3 different ways to access a block's elements.
  1. The nested types iterator and const_iterator. The range is [A.begin(), A.end()).
  2. The reverse_iterator and const_reverse_iterator are reverse iterator types. The range is [A.rbegin(), A.rend()).
  3. If n is an integer, the expression A[n] returns the n^{th} element. The expression A[n] is a shorthand for A.begin()[n], or, for that matter, for *(A.begin() + n).

Container Concepts (Cont.)

- The Hierarchy of Containers

Container
(Input Iterator)

↓

Forward Container
(Forward Iterator)

↓

Reversible Container
(Bidirectional Iterator)

↓

Random Access Container
(Random Access Iterator)

The Trivial Container

- ```
  template <class T>
  struct trivial_container {
     typedef T value_type;

     typedef value_type* pointer;
     typedef const value_type* const_pointer;
     typedef value_type& reference;
     typedef const value_type& const_reference;

     typedef value_type* iterator;
     typedef const value_type* const_iterator;

     typedef ptrdiff_t difference_type;
     typedef size_t size_type;

     const_iterator begin() const {  return 0; }
     const_iterator end() const { return 0; }


     iterator begin() {  return 0; }
     iterator end() { return 0; }

     size_type size() const { return 0; }
     bool empty() const { return 0; }
     size_type max_size() const { return 0; }

     void swap(trivial_container&) { }
  };
  ```
  ```

Variable Size Container Concepts

- The Container concept (along with Forward Container, Reversible Container, and Random Access Container) allows for the possibility of both fixed-size and variable size containers, but it can't provide any mechanism for adding elements to or deleting them from a container.
- The STL provides two different kinds of variable size containers: Sequence and Associative Container.

Sequences

- Sequence, a refinement of Forward Container, is the most obvious kind of variable size container.
- A Sequence doesn't arrange its own elements in a prescribed order. Instead, it gives you the tools so that you can arrange its elements in whatever order you need.
- The member functions insert and erase are really all there is to Sequences.
  - S.erase(p) removes that element from S and destroys it.
  - S.insert(p, x) creates a new object, a copy of x, and inserts it into S immediately *before* the element that p points to. S: Sequence, p: an iterator
- Two important questions:
  - When you insert or erase elements, what happens to the Sequence's other elements? **Answer: It depends.**
  - What is the complexity of insert and erase? **Answer: It depends.**

Sequences (Cont.)

- Other forms of insert and erase
  - For a Sequence, insert and erase are overloaded member functions.
  - Example: V.insert(V.begin(), L.begin(), L.end());
    // V is a vector and L is a list.
- Insertion at the Front and Back
  - Back Insertion Sequence and Front Insertion Sequence.
  - front(), push_front(), and pop_front() for Front Insertion Sequences.
  - back(), push_back(), and pop_back() for Back Insertion Sequences.

Sequences (Cont.)

- Insertion versus Overwrite Semantics
  - One of the most common mistakes is:

```
int A[5] = {1, 2, 3, 4, 5};
vector<int> V;
copy(A, A+5, V.begin());   // *(V.begin()) = A[0];
```

  - You can do it as:

```
int A[5] = {1, 2, 3, 4, 5};
vector<int> V;
copy(A, A+5, back_inserter(V));
// insert_iterator, fornt_insert_iterator, and
// back_insert_iterator adaptors
```

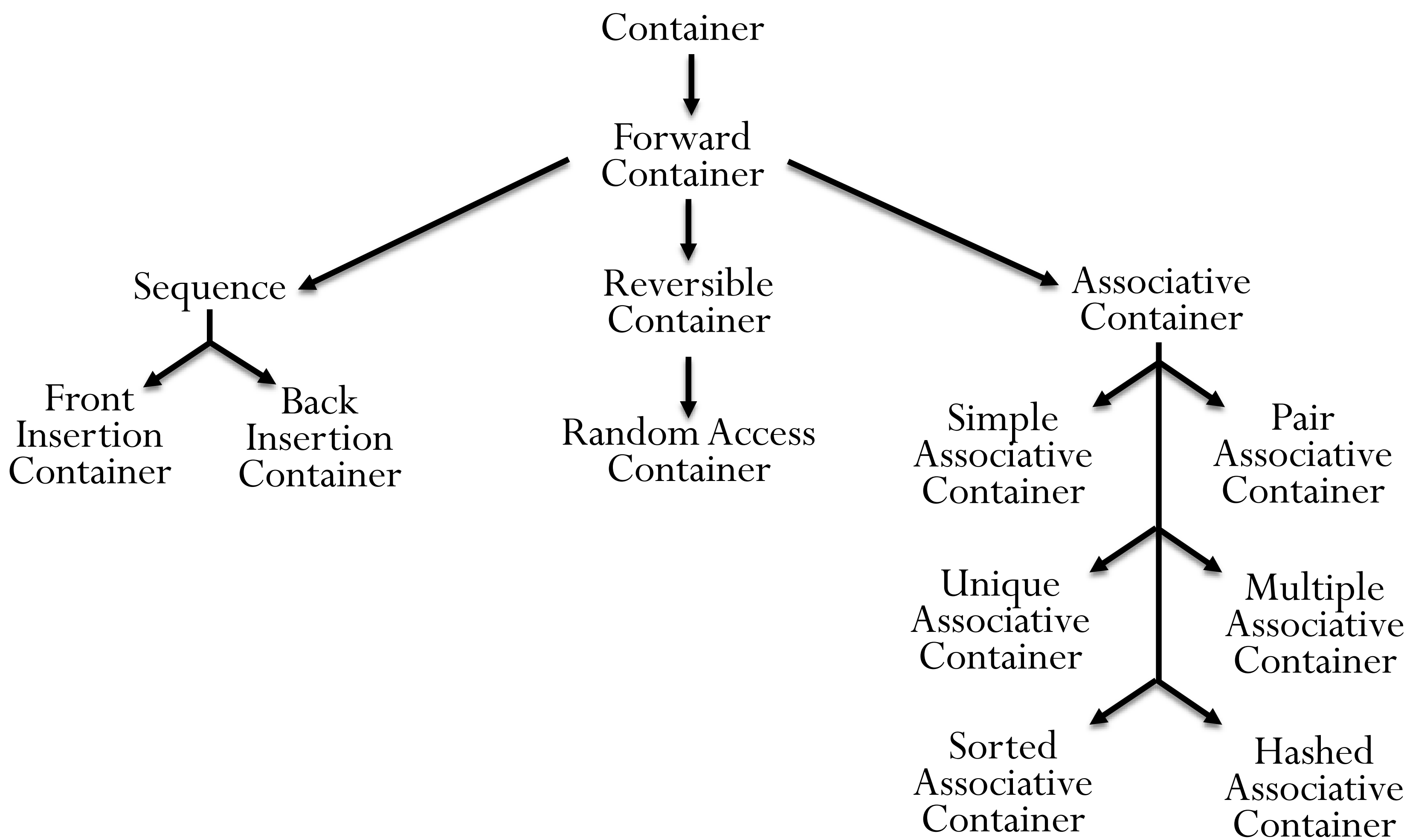
Associative Containers

- A Sequence does not impose a particular ordering: When you insert and element into a Sequence, you can insert it at any position.
- There's another kind of variable size container, one that guarantees that the elements are always ordering according to its own special rule.
  - You can look up an element faster. $O(N) \rightarrow O(\log N)$
- An Associative Container is a variable size container that supports efficient retrieval of elements (values) based on *keys*.
  - The basic operations are lookup, insertion, and erasure. Efficiency: $O(\log N)$ on average

Associative Containers (Cont.)

- Associative Containers are more complicated than Sequences.
  - Every element in an Associative Container has a key, and elements are looked up by their keys. It has value type and key type.
    - Simple Associative Container:
      value type and key type are the same type. It doesn't contain mutable iterators.
    - Pair Associative Container:
      value type is of the form pair<const Key, T>. Its key is the first pair's first field.
  - Can an Associative Container even contain two different elements with the same key?
    - Multiple Associative Container vs. Unique Associative Container
  - In every Associative Container, the elements are organized by key.
    - Hashed Associative Container:
      One of its nested type is a function object that is used as a hash function.
    - Sorted Associative Container:
      Its elements are always sorted in ascending order by key. It also has a nested type function object type, a Binary Predicate that is used to determine whether one key is less than another.

Summary



Container
Forward Container
Sequence
Reversible Container
Associative Container
Front Insertion Container
Back Insertion Container
Random Access Container
Simple Associative Container
Pair Associative Container
Unique Associative Container
Multiple Associative Container
Sorted Associative Container
Hashed Associative Container

Reference Manual – STL Concepts

Basic Concepts

Assignable

Default Constructable

Equality Comparable

Ordering

- LessThan Comparable

- Strick Weakly Comparable

Iterators (迭代器)

Trivial Iterator

Input Iterator

Output Iterator

Forward Iterator

Bidirectional Iterator

Random Access Iterator

Function Objects (函式物件)

Basic Function Objects

Adaptable Function Objects

Predicates

Specialized Concepts

Containers (容器)

General Container Concepts

Sequence Containers (序列式容器)

Associative Containers (關聯式容器)

Allocator (空間配置器)

Reference Manual:
Algorithms and Classes

Basic Components

pair

Iterator Primitives

allocator

Memory Management Primitives

Temporary Buffers

Nonmutating Algorithms
非變易演算法

Linear Search

Subsequence Matching

Counting Elements

for_each

Compare two Ranges

Maximum and Minimum Values

Basic Mutating Algorithms

Copy

Swapping Elements

transform

Replacing Elements

Filling Ranges

Removing Elements

Permuting Algorithms

Partitions

Random Shuffling and Sampling

Generalized Numeric Algorithms

Sorting and Searching

Sorting Ranges

Operations on Sorted Ranges

Heap Opertions

Iterator Classes

Insert Itrators

Stream Iterators

reverse_iterator

raw_storage_iterator

Function Object Classes

Function Object Base Classes

Arithmetic Operations

Comparisons

Logical Operations

Identity and Projection

Special Function Objects

Member Function Adaptors

The Other Adaptors

Container Classes

Sequences

Associative Containers (關聯式容器)

Container Adaptors

ConceptC++ Tutorial

Author: Douglas Gregor

Table of Contents

STL Containers

Container Operations (共通操作)

- Initialization (初始化)
  - Default Constructor
  - Copy Constructor
  - Destructor
  - Initializer List (初值列) (since C++11)
- Assignment and swap()
  - *move assignment* since C++11
- Size Operations
  - empty()
  - size()
  - max_size()

Container Operations (Cont.)

- Comparisons
  - ==, !=, <, <=, >, >=
- Element Access
  - range-based for iterations

```
for (auto& elem : coll) {
    std::cout << elem << std::endl;
}
for (auto pos=coll.begin(); pos!=coll.end(); ++pos) {
    *pos = …;
}
```

  - clear()
    - remove all elements in the container

Container Types

- size_type
- fifference_type
- value_type
- reference
- const_reference
- iterator
- const_iterator
- pointer
- const_pointer

Containers

- Sequence Containers
  - array, vector, deque, list, forward_list (since C++11)
  - implemented by dynamic arrays, linked lists
- Associative Containers
  - set, multiset, map, multimap
  - implemented by binary trees
- Unordered (associative) Containers
  - unordered_set, unordered_multiset, unordered_map, unordered_multimap
  - Implemented by hash tables

Sequence Containers - Arrays

- Abilities of Arrays
  - Initialization
  - swap() and Move Semantics
  - Size
- Array Operations
  - Create, Copy, and Destroy
  - Nonmodifying Operations
    - empty(), size(), max_size()
    - Comparisons
  - Assignments
    - fill(), swap()
  - Element Access
    - c[idx], c.at(idx), c.front(), c.back()
  - Iterator Functions
    - begin(), end(), cbegin(), cend(), rbegin(), rend(), crbegin(), crend()

Sequence Containers - Vectors

- Abilities of Vectors
  - Initialization
  - swap() and Move Semantics
  - Size and Capacity
- Vector Operations
  - Create, Copy, and Destroy
  - Nonmodifying Operations
    - empty(), size(), max_size(), capacity(), reserve()
    - Comparisons
  - Assignments
    - =, initlist, assign(), swap()
  - Element Access
    - c[idx], c.at(idx), c.front(), c.back()
  - Iterator Functions
    - begin(), end(), cbegin(), cend(), rbegin(), rend(), crbegin(), crend()
  - Inserting and Removing
    - push_back(), pop_back(), **insert(), emplace(), emplace_back(),** erase(), resize(), clear()
- Classs vector<bool>

Sequence Containers - Deques

- Abilities of Deques
  - Inserting and removing elements is fast both at the beginning and the end
  - shrink_to_fit() since C++11
- Deques Operations
  - Create, Copy, and Destroy
  - Nonmodifying Operations
    - empty(), size(), max_size() Note: It doesn't' provide capacity(), reserve()
    - Comparisons
  - Assignments
    - =, initlist, assign(), swap()
  - Element Access
    - c[idx], c.at(idx), c.front(), c.back()
  - Iterator Functions
  - Inserting and Removing
    - push_front(), pop_front(), push_back(), pop_back(), **insert(), emplace_back(), emblace_front()**, erase(), resize(), clear()

Sequence Containers - Lists

- Abilities of Lists
  - Two pointers (anchors) to the first element and the last elements
  - Each element has two pointers to the previous element and next element
  - It doesn’t provide random-access
- List Operations
  - Create, Copy, and Destroy
  - Nonmodifying Operations
  - Modifying Operations
    - **sort(), splice(), merge(), reverse(), unique(), swap()**
  - Assignments
  - Element Access
    - c.front(), c.back()
  - Iterator Functions
  - Inserting and Removing
    - push_front(), pop_front(), push_back(), pop_back(), insert(), emplace_back(), emblace_front(), erase(), resize(), clear()
    - remove(), remove_if()

Sequence Containers – Forward Lists

- Abilities of Forward Lists (since C++11)
  - It is a singly linked list and doesn't provide size().
- Forward List Operations
  - Create, Copy, and Destroy
  - Nonmodifying Operations
  - Assignments
  - Element Access
    - c.front()
  - Iterator Functions
    - before_begin(), cbefore_begin()
  - Inserting and Removing
    - push_front(), pop_front(), **insert_after()**, emplace_after(), emblace_front(), **erase_after()**, resize(), clear()
    - **remove(), remove_if()**
  - **Find and Remove or Insert**
  - **Splice Functions and Functions to Change the Order of Elements**

STL Iterators

Iterator Categories

- Header Files for Iterators
  - <iterator>
- Iterator Categories
  - Output Iterators (write forward)
  - Input Iterators (read forward once)
  - Forward Iterators (read forward)
  - Bidirectional Iterators (read forward and backward)
  - Random-Access Iterators (read with random access)

Auxiliary Iterator Functions

- advance()
- next() and prev()
- distance()
- iter_swap()

Iterator Adaptors

- Reverse Iterators
- Insert Iterators
  - Back Inserters
  - Front Inserters
  - General Inserters
- Stream Iterators
  - Ostream Iterators
  - Istream Iterators
- Move Iterators (since C++11)

Iterator Traits

- Writing Generic Functions for Iterators

User-Defined Iterators

STL Algorithms

Header Files

- #include <algorithm>
- #include <numeric>
- #include <functional>

Classification of Algorithms

- Nonmodifying Algorithms (非更易型)
- Modifying Algorithms
- Removing Algorithms
- Mutating Algorithms (變序型)
- Sorting Algorithms
- Sorted-Range Algorithms (已序區間)
- Numeric Algorithms

The for_each Algorithm

Nonmodifying Algorithms

Counting Elements

- difference_type
  count (InputIterator *beg*, InputIterator *end*, const T& *value*)
- difference_type
  count_if (InputIterator *beg*, InputIterator *end*, UnaryPredicate *op*)

Minimum and Maximum

- ForwardIterator
  min_element (ForwardIterator *beg*, ForwardIterator *end*)
- ForwardIterator
  min_element (ForwardIterator *beg*, ForwardIterator *end,*
  CompFunc *op*)
- ForwardIterator
  max_element (ForwardIterator *beg*, ForwardIterator *end*)
- ForwardIterator
  max_element (ForwardIterator *beg*, ForwardIterator *end,*
  CompFunc *op*)
- pair<ForwardIterator, ForwardIterator>
  minmax_element (ForwardIterator *beg*, ForwardIterator *end*)
- pair<ForwardIterator, ForwardIterator>
  minmax_element (ForwardIterator *beg*, ForwardIterator *end,*
  CompFunc *op*)

Searching Elements

- Search First Matching Element
  - InputIterator
    find (InputIterator *beg*, InputIterator *end*, const T& *value*)
  - InputIterator
    find_if (InputIterator *beg*, InputIterator *end*,
    UnaryPredicate *op*)
  - InputIterator
    find_if _not (InputIterator *beg*, InputIterator *end*,
    UnaryPredicate *op*)
- Search First n Matching Consecutive Elements
  - ForwardIterator
    search_n (ForwardIterator *beg*, ForwardIterator *end*, Size *count*,
    const T& *value*)
  - ForwardIterator
    search_n (ForwardIterator *beg*, ForwardIterator *end*, Size *count*,
    const T& *value*, BinaryPredicate *op*)

Searching Elements (Cont.)

- Search First Subrange
  - ForwardIterator1
    search (ForwardIterator1 *beg*, ForwardIterator1 *end*,
    ForwardIterator2 *searchBeg*, ForwardIterator2 *searchEnd*)
  - ForwardIterator1
    search (ForwardIterator1 *beg*, ForwardIterator1 *end*,
    ForwardIterator2 *searchBeg*, ForwardIterator2 *searchEnd*,
    BinaryPredicate *op*)

Searching Elements (Cont.)

- Search Last Subrange
  - ForwardIterator1 find_end (ForwardIterator1 *beg*, ForwardIterator1 *end*,
    ForwardIterator2 *searchBeg*, ForwardIterator2 *searchEnd*)
  - ForwardIterator1 find_end (ForwardIterator1 *beg*, ForwardIterator1 *end*,
    ForwardIterator2 *searchBeg*, ForwardIterator2 *searchEnd*,
    BinaryPredicate *op*)

Searching Elements (Cont.)

- Search First of Several Possible Elements
  - InputIterator
    find_first_of (InputIterator1 *beg*, InputIterator1 *end*,
    ForwardIterator2 *searchBeg*, ForwardIterator2
    searchEnd,
    BinaryPredicate *op*)
- Search Two Adjacent, Equal Elements
  - ForwardIterator
    adjacent_find (ForwardIterator *beg*, ForwardIterator *end*)
  - ForwardIterator
    adjacent_find (Forwardterator *beg*, ForwardIterator *end*,
    BinaryPredicate *op*)

Comparing Ranges

- Testing Equality
  - bool
    equal (InputIterator1 *beg*, InputIterator *end*,
    InputIterator2 *comBeg*)
  - bool
    equal (InputIterator1 *beg*, InputIterator *end*,
    InputIterator2 *comBeg,*
    BinaryPredicate *op*)

Comparing Ranges (Cont.)

- Testing for Unordered Equality
  - bool is_permutation (ForwardIterator1 *beg1*, ForwardIterator1 *end1*,
    ForwardIterator2 *beg2*)
  - bool is_permutation (ForwardIterator1 *beg1*, ForwardIterator1 *end1*,
    ForwardIterator2 *beg2,*
    CompFunc *op*)

Comparing Ranges (Cont.)

- Search First Difference
  - bool
    mismatch (InputIterator1 *beg1*, InputIterator1 *end1*,
    InputIterator2 *compBeg*)
  - bool
    mismatch (InputIterator1 *beg1*, InputIterator1 *end1*,
    InputIterator2 *compBeg,*
    BinaryPredicate *op*)

Comparing Ranges (Cont.)

- Testing for "Less Than"
  - bool lexicographical_compare (InputIterator1 *beg1*, InputIterator1 *end1*, InputIterator2 *beg2*, InputIterator2 *end2*)
  - bool lexicographical_compare (InputIterator1 *beg1*, InputIterator1 *end1*, InputIterator2 *beg2*, InputIterator2 *end2*, CompFunc *op*)

Predicates for Ranges

- Check for (Partial) Sorting
  - bool
    is_sorted (ForwardIterator *beg*, ForwarIterator *end*)
  - bool
    is_sorted (ForwardIterator *beg*, ForwardIterator *end*,
    BinaryPredicate *op*)
  - bool
    is_sorted_until (ForwardIterator *beg*, ForwardIterator *end*)
  - bool
    is_sorted_until (ForwardIterator *beg*, ForwardIterator *end*,
    BinaryPredicate *op*)

Predicates for Ranges (Cont.)

- Check for being Partitioned
  - bool
    is_partitioned (ForwardIterator *beg*, ForwarIterator *end*,
    UnaryPredicate *op*)
  - ForwardIterator
    partition_point (ForwardIterator *beg*, ForwardIterator *end*,
    BinaryPredicate *op*)

Predicates for Ranges (Cont.)

- Check for Being a Heap (Maximum Element First)
  - bool
    is_heap (RandomAccessIterator *beg*, RandomAccessIterator *end*)
  - bool
    is_heap (RandomAccessIterator *beg*, RandomAccess Iterator *end*,
    BinaryPredicate *op*)
  - RandomAccessIterator
    is_heap_until (RandomAccessIterator *beg*,
    RandomAccessIterator *end*)
  - RandomAccessIterator
    is_heap_until (RandomAccessIterator *beg*,
    RandomAccess Iterator *end*,
    BinaryPredicate *op*)

Predicates for Ranges (Cont.)

- All, Any, or None
  - bool
    all_of (InputIterator *beg*, InputIterator Iterator *end*,
    UnaryPredicate *op*)
  - bool
    any_of (InputIterator *beg*, InputIterator Iterator *end*,
    UnaryPredicate *op*)
  - bool
    none_of (InputIterator *beg*, InputIterator Iterator *end*,
    UnaryPredicate *op*)

References

- Generic Programming
  - http://www.generic-programming.org/
- Generic Programming in C++
  - http://www.generic-programming.org/languages/cpp/
- Standard Template Library Programmer's Guide
  - http://www.sgi.com/tech/stl/
- A Generic Programming Concept
  - http://www.cs.rpi.edu/~musser/concept-web/

Online Resources for the STL

- 侯捷觀點【Genericity/STL 大系】
- Standard Template Library Programmer's Guide - SGI
- C++ STL Tutorial - TutorialsPoint
- An Introduction to the C++ Standard Template Library (STL)
- C++ STL Tutorial - YoLinux.com
- The Tenouk's C++ Standard Template Library (STL)
- Standard Template Library (STL) Video Tutorials
- Alexander Stepanov: STL and Its Design Principles
- C++ Programming - Bo Qian's Space

Online Resources

- Nicolai M. Josuttis: The C++ Standard Library, 2nd edition
- Code Examples of the C++ Standard Library
- The C++ Standard Template Library (STL) Sample Codes Index Page
- Tutorialspoint
- cplusplus.com
- C++ and Standard Template Library (STL)

Online Resources (Cont.)

- **Programming -- Principles and Practice Using C++ (Second Edition)**
- **Welcome to Bjarne Stroustrup's homepage!**

Generic Programming

- PowerShow.com
  - Generic Programming
  - Templates and Generic Programming
  - Generic Programming Starts with *Algorithms*
- SlideShare.net
  - Generic Programming: The Good, the Bad, and the …

C++ Standard Template Library

- slideshare.net
  - Slides by llio Catallo – Politechnico di Milano

Software Test (軟體測試)

- 軟體工程 – 軟體測試
- 談軟體測試 - 悅知文化
- 軟體工程 - 測試策略
- **軟體測試專案實作：技術、流程與管理》筆記**

STL Source Documentation

- **libstdc++ Source Documentation**
- **Apache C++ Standard Library Reference Guide**
- The Microsoft STL

Visual Studio 2015

- Visual C++ in Visual Studio 2015
- C++ Language Reference

YouTube Videos

- AVL Trees Tutorial
  - https://www.youtube.com/watch?v=YKt1kquKScY
- Red-Black Trees
  - https://www.youtube.com/user/mikeysambol/videos
- Hash Tables
  - https://www.youtube.com/playlist?list=PLTZbNwgO5ebqw1v0ODk8cPLW9dQ99Te8f
  - https://www.youtube.com/watch?v=KyUTuwz_b7Q
- B-Tree Tutorial - An Introduction to B-Trees

C++ Regular Expressions

- C++ 11 Library: Regular Expression 1
- C++ 11 Library: Regular Expression 2 -- Submatch
- C++ 11 Library: Regular Expression 3 -- Iterators
- C++ Introduction to Regular Expression (Using Regex Library)
- CppCon 2016: Tim Shen "Regular Expressions in C++, Present and Future"

JavaScript Regular Expressions

- 2.1: Introduction to Regular Expressions - Programming with Text
- 2.2: Regular Expressions: Meta-characters – Programming with Text
- 2.3: Regular Expressions: Character Classes – Programming with Text
- 2.4: Regular Expressions: Capturing Groups – Programming with Text
- 2.5: Regular Expressions: Back References – Programming with Text

JavaScript Regular Expressions

- 2.6: Regular Expressions: test() and match() – Programming with Text
- 2.7: Regular Expressions: exec() – Programming with Text
- 2.8: Regular Expressions: split() – Programming with Text
- 2.9: Regular Expressions: replace() – Programming with Text

C++ Tutorials

- **Tommy Ngo**
- Bo Qian
- The Coding Train
- Jason Turner
- **C++ Tutorial for Beginners**
- **C++ Tutorials From Basic to Advance**
- **C++ Programming**
- **C++ Tutorial | Learn C++ programming**

Modern C++

- A Tour of Modern C++
- Modern C++: What You Need to Know
- CppCon 2017: Jason Turner "Practical C++17"
- CppCon 2017: Bjarne Stroustrup "Learning and Teaching Modern C++"

CppCon

- CppCon
- Meeting Cpp
- CppCon 2016: Bjarne Stroustrup "The Evolution of C++ Past, Present and Future"

C++ Unit Testing

- C++ Unit Testing with Google Test Tutorial

JetBrains TV

- JetBrainsTV

Woboq + libstdc++

- https://code.woboq.org/
- **libstdc++ Source Documentation**
- gcc/libstdc++-v3 at master · gcc-mirror/gcc · GitHub

Object-Oriented Software Design and Construction with C++

- http://www.prenhall.com/divisions/esm/app/kafura/secure/toc.htm
- http://www.java2s.com/Tutorial/
- http://www.java2s.com/Tutorial/Cpp/CatalogCpp.htm