Object-Oriented Programming and Data Structures

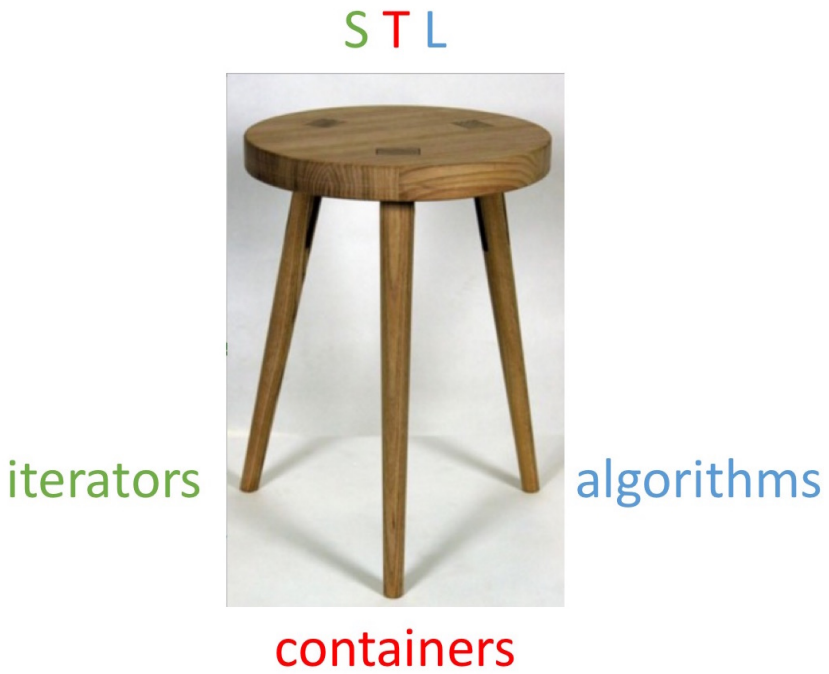
COMP2012: Standard Template Library (STL) for Generic Programming

Brian Mak
Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

The Standard Template Library (STL)

- The STL is a collection of powerful, template-based, reusable codes.
- It implements many general-purpose containers (data structures) together with algorithms that work on them.
- To use the STL, we need an understanding of the following 3 topics:



Part I

STL Containers

Container Classes

- A container class is a class that holds a collection of homogeneous objects — of the same type.
- Container classes are a typical use of class templates since we frequently need containers for homogeneous objects of different types at different times.
- The object types need not be known when the container class is designed.
- Let's design a sequence container that looks like an array, but that is a first-class type: so assignment and call by value is possible.
- Remark: The vector class in STL is better; so this is just an exercise for your understanding.

An Array Container Class

```
1  template <typename T>   /* File: arrayT.h */
2  class Array
3  {
4    private:
5      int _size;
6      T* _value;
7
8    public:
9      Array<T>(int n = 10);   // Default and conversion constructor
10     Array<T>(const Array<T>&); // Copy constructor
11     ~Array<T>();
12
13     int size() const { return _size; }
14     void init(const T& k);
15
16     Array<T>& operator=(const Array<T>& a);// Copy assignment operator
17     T& operator[](int i) { return _value[i]; } // lvalue
18     const T& operator[](int i) const { return _value[i]; } // rvalue
19 };
```

An Array Container Class Too

Within the template, the typename for Array may be omitted.

```
1  template <typename T>   /* File: array.h */
2  class Array
3  {
4    private:
5      T* _value;
6      int _size;
7
8    public:
9      Array(int n = 10);   // Default and conversion constructor
10     Array(const Array&); // Copy constructor
11     ~Array();
12
13     int size() const { return _size; }
14     void init(const T& k);
15
16     Array& operator=(const Array&);      // Copy assignment operator
17     T& operator[](int i) { return _value[i]; } // lvalue
18     const T& operator[](int i) const { return _value[i]; } // rvalue
19 };
```

Example: Use of Class Array

```
1   #include <iostream>     /* File: test-array.cpp */
2   using namespace std;
3   #include "array.h"
4   #include "array-constructors.h"
5   #include "array-op=.h"
6   #include "array-op-os.h"
7
8   int main()
9   {
10      Array<int> a(3);
11      a.init(98); cout << a << endl;
12      a = a; a[2] = 17; cout << a << endl;
13
14      Array<char> b(4);
15      b.init('g'); b[0] = a[1]; cout << b << endl;
16
17      const Array<char> c = b;
18      // c[2] = 5; // Error: assignment of read-only location
19      cout << c << endl;
20
21      Array<int> d;
22      d = a; cout << d << endl;
23      return 0;
24  }
```

Constructors/Destructor of Class Array

```
1  template <typename T>   /* File: array-constructors.h */
2  Array<T>::Array(int n) : _value( new T [n] ), _size(n) { }
3
4  template <typename T> Array<T>::Array(const Array<T>& a)
5      : Array(a._size)   // Delegating constructor
6  {
7      for (int i = 0; i < _size; ++i)
8          _value[i] = a._value[i];
9  }
10
11 template <typename T> Array<T>::~Array() { delete [] _value; }
12
13 template <typename T> void Array<T>::init(const T& k)
14 {
15     for (int i = 0; i < _size; ++i)
16         _value[i] = k;
17 }
```

Assignment Operators of Class Array: Deep/Shallow Copy

```
1  template <typename T>     /* File: array-op=.h */
2  Array<T>& Array<T>::operator=(const Array<T>& a) // Deep copy
3  {
4      if (&a != this)       // Avoid self-assignment: e.g., a = a
5      {
6          delete [] _value;             // First remove the old data
7          _size = a._size;
8          _value = new T [_size];       // Re-allocate memory
9
10         for (int j = 0; j <_size; ++j) // Copy the new data
11             _value[j] = a[j];
12     }
13
14     return (*this);
15 }
```

Non-member Operator≪ as a Global Function Template

- Function templates and class templates work together very well: We can use function templates to implement functions that will work on any class created from a class template.

```
1  template <typename T>   /* File: array-op-os.h */
2  ostream& operator<<(ostream& os, const Array<T>& a)
3  {
4      os << "#elements stored = " << a.size() << endl;
5
6      for (int j = 0; j < a.size(); ++j)
7          os << a[j] << endl;
8
9      return os;
10 }
```

Operator≪ as a Friend Function Template

- The Array class template may declare the operator≪ as a friend function inside its definition as a function template.

```
1  template <typename T>   /* File: array-w-os-friend.h */
2  class Array
3  {
4      template <typename S>
5          friend ostream& operator<<(ostream& os, const Array<S>& x);
6
7    private:
8      T* _value;
9      int _size;
10
11   public:
12     Array(int n = 10);   // Default and conversion constructor
13     Array(const Array&); // Copy constructor
14     ~Array();
15
16     int size() const { return _size; }
17     void init(const T& k);
18     Array& operator=(const Array&);      // Copy assignment operator
19     T& operator[](int i) { return _value[i]; } // lvalue
20     const T& operator[](int i) const { return _value[i]; } // rvalue
21 };
```

Operator≪ as a Friend Function Template ..

- The friend operator≪ function definition may be defined outside the Array class template like other class member functions.
- Now the friend operator≪ function may access the private members of the Array class.

```
1  template <typename T>   /* File: array-op-os-friend.h */
2  ostream& operator<<(ostream& os, const Array<T>& a)
3  {
4      os << "#elements stored = " << a._size << endl;
5
6      for (int i = 0; i < a._size; ++i)
7          os << a._value[i] << endl;
8
9      return os;
10 }
```

Containers in STL

Type of Container	STL Containers
Sequence	vector, list, deque
Associative	map, multimap, multiset, set
Adaptors	priority_queue, queue, stack
Near-containers	bitset, valarray, string

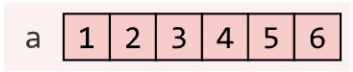
- Containers in the same category share a set of same or similar public member functions (i.e., public interface or algorithms).

Containers in STL ..

1. Sequence containers
   - Represent sequential data structures
   - Start from index/location 0
2. Associative containers
   - Non-sequential containers
   - Store (key, value) pairs
3. Container adaptors
   - adapted containers that support a limited set of container operations
4. "Near-containers" C-like pointer-based arrays
   - Exhibit capabilities similar to those of the sequence containers, but do not support all their capabilities
   - strings, bitsets and valarrays

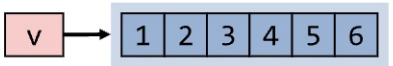
Examples: STL Sequence Container Classes

`array<T,size>`



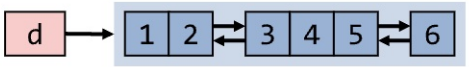
fixed-size contiguous array

`vector<T>`



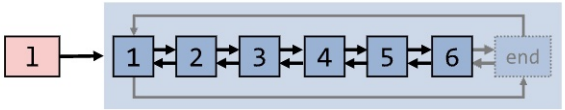
dynamic contiguous array; amortized $O(1)$ growth strategy;
C++'s "default" container

`deque<T>`



double-ended queue; fast insert/erase at both ends

`list<T>`



doubly-linked list; $O(1)$ insert, erase & splicing; in practice often slower than vector

Some Properties of STL Sequence Containers

Container	Access Control	Add/Remove
vector (1D array)	O(1) random access	O(1) at the end O(n) in front/middle
list (doubly-linked list)	O(n) in the middle O(1) at front/end	O(1) at any position
deque (double-ended queue)	O(1) random access	O(1) at front/back O(n) in the middle

Sequence Containers: Access, Add, Remove

Element access for all:

- front( ): First element
- back( ): Last element

Element access for vector and deque:

- [ ]: Subscript operator, index not checked.

Add/remove elements for all:

- push_back( ): Append element.
- pop_back( ): Remove last element.

Add/remove elements for list and deque:

- push_front( ): Insert element at the front.
- pop_front( ): Remove first element.

Sequence Containers: Other Operations

List operations are fast for list, but also available for vector and deque:

- insert(p, x): Insert an element **x** at position **p**.
- erase(p): Remove an element at position **p**.
- clear( ): Erase all elements.

Miscellaneous Operations:

- size( ): Return the number of elements.
- empty( ): Return true if the sequence is empty.
- resize(int new_size): Change size of the sequence.

Comparison operators ==, !=, < etc. are also defined.

Part II

Container Adaptors: Stack and Queue

Stack: How it Works

Consider a pile of cookies.

- more cookies: new cookies are added on top, one at a time.
- fewer cookies: cookies are consumed one at a time, starting at the top.

As a container adaptor, insertions and removals of items on a stack are based on the *last-in first-out (LIFO)* policy.

It supports:

- Data: an ordered list of data/items.
- Operations (major ones):

  top : get the value of the top item

  push : add a new item to the top

  pop : remove an item from the top

Simplified STL Stack

- typedef is a keyword used to introduce a synonym for an existing type expression:

  typedef <a type expression> <type-synonym>

```
1  template<typename T, typename Sequence = deque<T> >
2  class stack
3  {
4    protected:
5      Sequence c; // Underlying container
6
7    public:
8      typedef typename Sequence::value_type         value_type;
9      typedef typename Sequence::reference          reference;
10     typedef typename Sequence::const_reference    const_reference;
11     typedef typename Sequence::size_type          size_type;
12
13     // (Default) Constructor
14     explicit stack(const Sequence& _c = Sequence()) : c(_c) { }
```

Simplified STL Stack ..

```
15
16      // Return true if the stack is empty
17      bool empty() const { return c.empty(); }
18
19      // Return the number of elements in the stack
20      size_type size() const { return c.size(); }
21
22      // Return a R/W reference to the data at the first element
23      reference top() { return c.back(); }
24
25      // Read-only version of top()
26      const_reference top() const { return c.back(); }
27
28      // Create an element at the top of the stack and assign x to it
29      void push(const value_type& x) { c.push_back(x); }
30
31      // Shrink the stack by one. Note that no data is returned.
32      void pop() { c.pop_back(); }
33  };
```

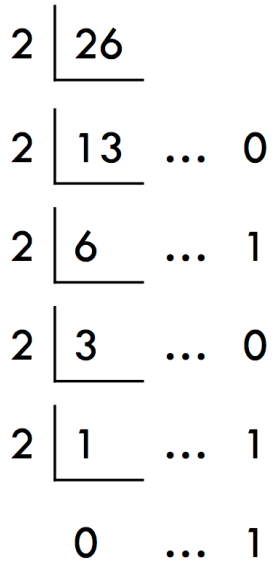
Example: Decimal to Binary Conversion — Illustration

- e.g., $26_{(10)} = 11010_{(2)}$
- Algorithm to convert $N_{(10)} = M_{(2)}$:

Step 1 : divide N by 2 successively

Step 2 : each time push the remainder onto a stack

Step 3 : print the answer by popping the stack successively



Example: Decimal to Binary Conversion

```
1   #include <iostream>     /* File: decimal2binary.cpp */
2   #include <stack>
3   using namespace std;
4
5   int main()  // Convert +ve decimal number to binary number using a stack
6   {
7       stack<int> a;
8       int x, number;
9
10      while (cin >> number)
11      {
12          // Conversion: decimal to binary
13          x = number;
14          do { a.push(x % 2); x /= 2; } while (x > 0);
15
16          // Print a binary that is stored on a stack
17          cout << number << " (base 10) = ";
18          while (!a.empty()) { cout << a.top(); a.pop(); }
19          cout << " (base 2)" << endl;
20      }
21
22      return 0;
23  }
```

Example: Balanced Parentheses — Illustration

- e.g., [()][()()]() is balanced but [(]) is not.
- Algorithm to check balanced parentheses:

Step 1 : Scan the given character expression from left to right.

Step 2 : If a left parenthesis is read, push it onto a stack.

Step 3 : If a right parenthesis is read, check if its matching left parenthesis is on the top of the stack.

Step 4 : If Step 3 is true, pop the stack and continue.

Step 5 : If Step 3 is false, return false and stop.

Step 6 : If the end of the expression is reached, check if the stack is empty.

Step 7 : If Step 6 is true, return true otherwise false.

Example: Balanced Parentheses

```
1   #include <iostream>     /* File: balanced-paren.cpp */
2   #include <stack>
3   using namespace std;
4
5   const char L_PAREN    = '('; const char R_PAREN    = ')';
6   const char L_BRACE    = '{'; const char R_BRACE    = '}';
7   const char L_BRACKET  = '['; const char R_BRACKET  = ']';
8   bool balanced_paren(const char* expr);
9
10  int main()  // To check if a string has balanced parentheses
11  {
12      char expr[1024];
13      cout << "Input an expression containing parentheses: ";
14      cin >> expr;
15      cout << boolalpha << balanced_paren(expr) << endl;
16      return 0;
17  }
18
19  bool check_char_stack(stack<char>& a, char c)
20  {
21      if (a.empty()) return false;
22      if (a.top() != c) return false;
23      a.pop(); return true;
24  }
```

Example: Balanced Parentheses ..

```
25  bool balanced_paren(const char* expr)
26  {
27      stack<char> a;
28      for (const char* s = expr; *s != '\0'; ++s)
29          switch (*s)
30          {
31              case L_PAREN: case L_BRACE: case L_BRACKET:
32                  a.push(*s); break;
33
34              case R_PAREN:
35                  if (!check_char_stack(a, L_PAREN)) return false;
36                  break;
37              case R_BRACE:
38                  if (!check_char_stack(a, L_BRACE)) return false;
39                  break;
40              case R_BRACKET:
41                  if (!check_char_stack(a, L_BRACKET)) return false;
42                  break;
43
44              default: break;
45          }
46
47      return a.empty();
48  }
```

Queue: How it Works

Consider the case when people line up for tickets.

- more people: new customers join the back of a queue, one at a time.
- fewer people: the customer at the front buys a ticket and leaves the queue.

As a container adaptor, insertions and removals of items on a queue are based on a *first-in first-out (FIFO)* policy.

It supports:

- Data: an ordered list of data/items.
- Operations (major ones):

front : get the value of the front item

enqueue : add a new item to the back

dequeue : remove an item from the front

Simplified STL Queue

```
1  template<typename T, typename Sequence = deque<T> >
2  class queue
3  {
4    protected:
5      Sequence c; // Underlying container
6
7    public:
8      typedef typename Sequence::value_type         value_type;
9      typedef typename Sequence::reference          reference;
10     typedef typename Sequence::const_reference    const_reference;
11     typedef typename Sequence::size_type          size_type;
12
13     // (Default) Constructor
14     explicit queue(const Sequence& _c = Sequence()) : c(_c) { }
15
16     // Return true if the queue is empty
17     bool empty() const { return c.empty(); }
18
19     // Return the number of elements in the queue
20     size_type size() const { return c.size(); }
21
22     // Return a R/W reference to the data at the first element of the queue
23     reference front() { return c.front(); }
```

Simplified STL Queue ..

```
24
25      // Read-only version of front()
26      const_reference front() const { return c.front(); }
27
28      // Return a R/W reference to the data at the last element of the queue
29      reference back() { return c.back(); }
30
31      // Read-only version of back()
32      const_reference back() const { return c.back(); }
33
34      // Create an element at the end of the queue and assigns x to it
35      // i.e., enqueue
36      void push(const value_type& x) { c.push_back(x); }
37
38      //  It shrinks the queue by one. Note that no data is returned.
39      // i.e., dequeue
40      void pop() { c.pop_front(); }
41  };
```

Example: Queue of int Data

```
1   #include <iostream>     /* File: int-queue-test.cpp */
2   #include <queue>
3   using namespace std;
4
5   void print_queue_info(const queue<int>& a) {
6       cout << "\nNo. of data currently on the queue = " << a.size() << endl;
7       if (!a.empty()) {
8           cout << "First: " << a.front() << "\nLast: " << a.back() << endl; }
9   }
10  int main()
11  {
12      queue<int> a; print_queue_info(a);
13      a.push(4);    print_queue_info(a);
14      a.push(15);   print_queue_info(a);
15      a.push(26);   print_queue_info(a);
16      a.push(37);   print_queue_info(a);
17      a.pop();      print_queue_info(a);
18      a.push(48);   print_queue_info(a);
19      a.push(59);   print_queue_info(a);
20      a.pop();      print_queue_info(a);
21      a.pop();      print_queue_info(a);
22      a.pop();      print_queue_info(a);
23      a.pop();      print_queue_info(a);
24      a.pop();      print_queue_info(a); return 0;
25  }
```

Part III

STL Iterators: Generalized Pointers

Pointers to Traverse an Array of a Basic Type

```
1   #include <iostream>     /* File: print-int-array.cpp */
2   using namespace std;
3
4   int main()
5   {
6       const int LENGTH = 5;
7       int x[LENGTH];
8
9       for (int j = 0; j < LENGTH; ++j)
10          x[j] = j;
11
12      // x_end points to a non-existing element just beyond the array
13      const int* x_end = &x[LENGTH];
14
15      for (const int* p = x; p != x_end; ++p)
16          cout << *p << endl;
17
18      return 0;
19  }
```

Pointers to Traverse an Array of a Basic Type ..

- For a sequence of values of basic types, one may set up a pointer, **p**, of the type which supports the following operations:

Operation	Goal
`p = x`	Initialize to the beginning of an array
`*p`	Access an element by dereferencing its pointer
`p→`	Access an element pointed to by its pointer
`--p`	To point to the previous element
`++p`	To point to the next element
`==, !=`	Pointer comparisons

Iterators to Traverse a Sequence Container

- Iterators are generalized pointers.
- To traverse the elements of a sequence container sequentially, one may use an iterator of the container type. E.g, list<int>::iterator is an iterator for a **list** of **int**.
- const_iterator is the const version of an iterator: the object it 'points' to can't be modified.
- STL sequence containers provide the begin( ) and end( ) to set an iterator to the beginning and end of a container.
- For each kind of STL sequence container, there is an iterator type. E.g.,
  - list<int>::iterator, list<int>::const_iterator
  - vector<string>::iterator, vector<string>::const_iterator
  - deque<double>::iterator, deque<double>::const_iterator

Iterators to Traverse a Sequence Container ..

```
1   #include <iostream>          /* File: print-list.cpp */
2   using namespace std;
3   #include <list>              // STL list
4
5   int main()
6   {
7       list<int> x;                    // An int STL list
8       for (int j = 0; j < 5; ++j)
9           x.push_back(j);             // Append items to an STL list
10
11      list<int>::const_iterator p; // STL list iterator
12      for (p = x.begin(); p != x.end(); ++p)
13          cout << *p << endl;
14
15      return 0;
16  }
```

Example: find( ) With an int Iterator

- Iterator provides a common interface to access elements of a sequence container without making any difference between different container classes.
- The same code works for all sequence container classes.

```
1  typedef int* Int_Iterator; /* File: find-int-iterator.cpp */
2
3  /* Actually this find function is already defined in STL */
4  Int_Iterator
5  find(Int_Iterator begin, Int_Iterator end, const int& value)
6  {
7      while (begin != end && *begin != value)
8          ++begin;
9
10     return begin;
11 }
```

Example: find( ) With an int Iterator . . .

```
1   #include <iostream>     /* File: find-test.cpp */
2   using namespace std;
3   typedef int* Int_Iterator;
4
5   int main()
6   {
7       const int SIZE = 10; int x[SIZE];
8       for (int i = 0; i < SIZE; i++)
9           x[i] = 2 * i;
10
11      Int_Iterator begin = x; Int_Iterator end = &x[SIZE];
12      while (true)
13      {
14          cout << "Enter number: "; int num; cin >> num;
15          Int_Iterator position = find(begin, end, num);
16
17          if (position == end)
18              cout << "Not found\n";
19          else if (++position != end)
20              cout << "Found before the item " << *position << '\n';
21          else
22              cout << "Found as the last element\n";
23      }
24      return 0;
25  }
```

Why Are Iterators So Great?

```
1  template <class Iterator, class T> /* File: find-template.h */
2  Iterator find(Iterator begin, Iterator end, const T& value)
3  {
4      while (begin != end && *begin != value)
5          ++begin;
6
7      return begin;
8  }
```

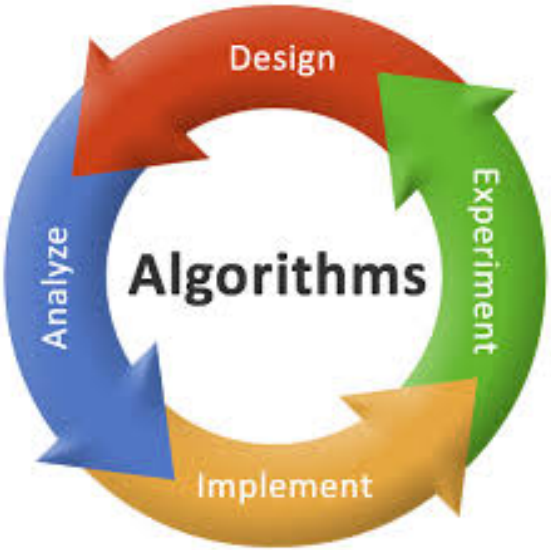
- Iterators allow us to separate algorithms from containers when they are used with templates.
- The new **find( )** function template contains no information about the implementation of the container, or how to move the iterator from one element to the next.
- The same **find**( ) function can be used for any container that provides a suitable iterator.

Example: find( ) with a vector Iterator

```
1   #include <iostream>     /* File: find-iterator-test.cpp */
2   using namespace std;
3   #include <vector>
4
5   int main()
6   {
7       const int SIZE = 10; vector<int> x(SIZE);
8       for (int i = 0; i < x.size(); i++)
9           x[i] = 2 * i;
10
11      while (true)
12      {
13          cout << "Enter number: "; int num; cin >> num;
14          vector<int>::iterator position = find(x.begin(), x.end(), num);
15
16          if (position == x.end())
17              cout << "Not found\n";
18          else if (++position != x.end())
19              cout << "Found before the item " << *position << '\n';
20          else
21              cout << "Found as the last element\n";
22      }
23
24      return 0;
25  }
```

Part IV

STL Algorithms



STL Algorithms

- The STL does not only have container classes and iterators, but also algorithms that work with different containers.
- STL algorithms are implemented as global functions.
- E.g., STL algorithm find( ) searches sequentially through a sequence, and stops when an item matches its 3rd argument.
- One limitation of find( ) is that it requires an exact match by value.

```
1  template <class Iterator, class T> /* File: stl-find.cpp */
2  Iterator find(Iterator first, Iterator last, const T& value)
3  {
4      while (first != last && *first != value)
5          ++first;
6
7      return first;
8  }
```

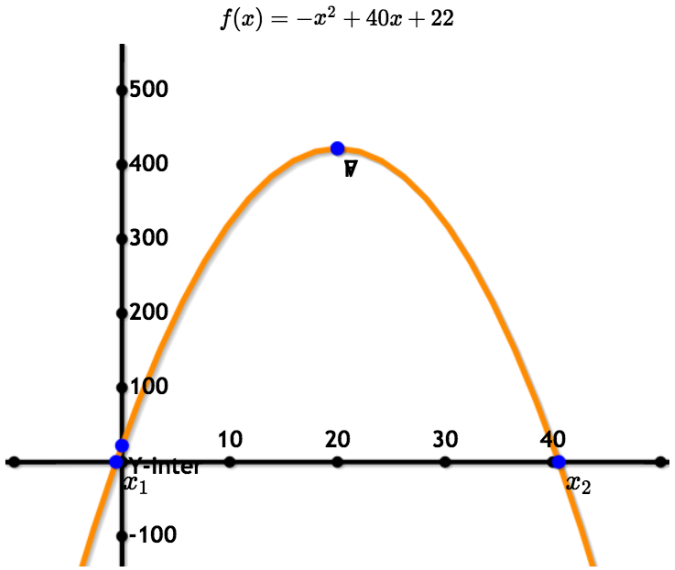
Example: Using STL find( )

```
1   #include <iostream>     /* File: find-composer.cpp */
2   using namespace std;
3   #include <string>
4   #include <list>
5   #include <algorithm>
6
7   int main()
8   {
9       list<string> composers;
10      composers.push_back("Mozart");
11      composers.push_back("Bach");
12      composers.push_back("Chopin");
13      list<string>::iterator p =
14          find(composers.begin(), composers.end(), "Bach");
15
16      if (p == composers.end())
17          cout << "Not found." << endl;
18      else if (++p != composers.end())
19          cout << "Found before: " << *p << endl;
20      else
21          cout << "Found at the end of the list." << endl;
22
23      return 0;
24  }
```

Algorithms, Iterators, and Sub-Sequences

Sequences/Sub-sequences are specified using iterators that indicate the beginning and the end for an algorithm to work on.

The following functions will be used in the following examples.




```
1  /* File: init.h */
2  inline int quadratic(int x) { return -x*x + 40*x + 22; }
3
4  template <typename T>
5  void my_initialization(T& x, int num_items)
6  {
7      for (int j = 0; j < num_items; ++j)
8          x.push_back( quadratic(j) ); // Can you rewrite using lambda?
9  }
```

Example: STL find( ) the 2nd Occurrence of a Value

```
1  #include <iostream>     /* File: find-2nd-occurrence.cpp */
2  using namespace std;
3  #include <vector>
4  #include <algorithm>
5  #include "init.h"
6
7  int main()
8  {
9      const int search_value = 341;
10     vector<int> x;
11     my_initialization(x, 100);
12
13     vector<int>::iterator p = find(x.begin(), x.end(), search_value);
14
15     if (p != x.end())    // Value found for the first time!
16     {
17         p = find(++p, x.end(), search_value); // Search again
18         if (p != x.end())
19             cout << search_value << " appears after " << *--p << endl;
20     }
21     return 0;
22 }
```

STL find_if( )

```
1  template <class Iterator, class Predicate> /* File: stl-find-if.cpp */
2  Iterator find_if(Iterator first, Iterator last, Predicate predicate)
3  {
4      while (first != last && !predicate(*first))
5          ++first;
6
7      return first;
8  }
```

- find_if( ) is a more general algorithm than find( ) in that it stops when a condition is satisfied.
- The condition is called a predicate and is implemented by a boolean function.
- This allows partial match, or match by keys.
- In general, you may pass a function to another function as its argument!

STL find_if( ) — Search by Condition

```
1  #include <iostream>     /* File: find-gt350.cpp */
2  using namespace std;
3  #include <vector>
4  #include <algorithm>
5  #include "init.h"
6
7  bool greater_than_350(int value) { return value > 350; }
8
9  int main()
10 {
11     vector<int> x;
12     my_initialization(x, 100);
13
14     vector<int>::const_iterator p =
15         find_if( x.begin(), x.end(), greater_than_350 );
16
17     if (p != x.end())
18         cout << "Found element: " << *p << endl;
19
20     return 0;
21 }
```

Function Pointer

- Inherited from C, C++ allows a function to be passed as argument to another function.
- Actually, we say that we pass the function pointer.
- E.g., the type of the function pointer of the template `larger()` we talked before is:

```
inline const T& (*)(const T&, const T&)
```

- STL's `max()` is the same as our `larger()`.

Example: Function Pointer — smaller( ) and larger( )

```
1   #include <iostream>      /* File: fp-smaller-larger.cpp */
2   using namespace std;
3
4   int larger(int x, int y) { return (x > y) ? x : y; }
5   int smaller(int x, int y) { return (x > y) ? y : x; }
6
7   int main()
8   {
9       int choice;
10      cout << "Choice: (1 for larger; others for smaller): ";
11      cin >> choice;
12
13      int (*f)(int, int) = (choice == 1) ? larger : smaller;
14
15      cout << f(3, 5) << endl;
16      return 0;
17  }
```

Example: Array of Function Pointers — Calculator

```
1  #include <iostream>     /* File: fp-calculator.cpp */
2  using namespace std;
3  double add(double x, double y) { return x+y; }
4  double subtract(double x, double y) { return x-y; }
5  double multiply(double x, double y) { return x*y; }
6  double divide(double x, double y) { return x/y; } // No error checking
7
8  int main()
9  {
10     double (*f[])(double x, double y) // Array of function pointers
11         = { add, subtract, multiply, divide };
12
13     int operation; double x, y;
14     cout << "Enter 0:+, 1:-, 2:*, 3:/, then 2 numbers: ";
15     while (cin >> operation >> x >> y)
16     {
17         if (operation >= 0 && operation <= 3)
18             cout << f[operation](x, y) << endl; // Call + - * /
19         cout << "Enter 0:+, 1:-, 2:*, 3:/, then 2 numbers: ";
20     }
21     return 0;
22 }
```

Example: Function Pointer as Lambda

```
1  #include <iostream>     /* File: fp-smaller-larger-lambda.cpp */
2  using namespace std;
3
4  int main()
5  {
6      int choice;
7      cout << "Choice: (1 for larger; others for smaller): ";
8      cin >> choice;
9
10     int (*f)(int, int);
11
12     if (choice == 1)
13         f = [] (int x, int y) { return (x > y) ? x : y; };
14     else
15         f = [] (int x, int y) { return (x > y) ? y : x; };
16
17     cout << f(3, 5) << endl;
18     return 0;
19 }
```

Function Objects

- STL function objects are a generalization of function pointers.
- An object that can be called like a function is called a function object, functoid, or functor.
- Function pointers and lambdas just two example of function objects.
- An object can be called if it supports operator( ).
- A function object must have at least operator( ) overloaded; of course, they may have other member functions/data.
- Function objects are more powerful than function pointers, since they can have data members and therefore carry around information or internal states.
- A function object (or a function) that returns a boolean value (of type bool) is called a predicate.

STL find_if( ) with Function Object Greater_Than

```
1   #include <iostream>     /* File: fo-greater-than.cpp */
2   using namespace std;
3   #include <algorithm>
4   #include <vector>
5   #include "init.h"
6   #include "fo-greater-than.h"
7
8   int main()
9   {
10      vector<int> x; my_initialization(x, 100);
11      int limit = 0;
12
13      while (cin >> limit)
14      {
15          vector<int>::const_iterator p =
16              find_if(x.begin(), x.end(), Greater_Than(limit)); // Call FO
17
18          if (p != x.end())
19              cout << "Element found: " << *p << endl;
20          else
21              cout << "Element not found!" << endl;
22      }
23
24      return 0;
25  }
```

STL find_if( ) with Function Object Greater_Than ..

```
1  class Greater_Than       /* File: fo-greater-than.h */
2  {
3    private:
4      int limit;
5    public:
6      Greater_Than(int a) : limit(a) { }
7      bool operator()(int value) { return value > limit; }
8  };
```

- The line with `Call FO` is the same as:

```
// Create a Greater_Than temporary function object g
Greater_Than g(350); // a temporary object
p = find_if( x.begin(), x.end(), g );
```

- When find_if( ) examines each item, say `x[j]` in the container vector<int> x, against the temporary Greater_Than function object, it will call the FO's operator( ) with `x[j]` as the argument. i.e.,
  `g(x[j]) // Or, in formal writing: g.operator()(x[j])`

STL count_if( ) with Function Object Greater_Than

```
1   #include <iostream>      /* File: fo-count.cpp */
2   using namespace std;
3   #include <vector>
4   #include <algorithm>
5   #include "fo-greater-than.h"
6
7   int main()
8   {
9       vector<int> x;
10      for (int j = -5; j < 5; ++j)
11          x.push_back(j*10);
12
13      // Count how many items are greater than 10
14      cout << count_if(x.begin(), x.end(), Greater_Than(10)) << endl;
15
16      return 0;
17  }
```

STL for_each( ) to Sum using Function Object

```
1  #include <iostream>     /* File: fo-sum.cpp */
2  using namespace std;
3  #include <list>
4  #include <algorithm>
5
6  class Sum
7  {
8    private:
9      int sum;
10   public:
11     Sum() : sum(0) { }
12     void operator()(int value) { sum += value; }
13     int result() const { return sum; }
14 };
15
16 int main()
17 {
18     list<int> x;
19     for (int j = 0; j < 5; ++j) x.push_back(j); // Initialize x
20     Sum sum = for_each( x.begin(), x.end(), Sum() );
21     cout << "Sum = " << sum.result() << endl; return 0;
22 }
```

STL Algorithms: for_each( ) and transform( )

```
1  /* File: stl-foreach.h */
2  template <class Iterator, class Function>
3  Function for_each(Iterator first, Iterator last, Function g)
4  {
5      for ( ; first != last; ++first )
6          g(*first);
7
8      return g; // Returning the input function!
9  }
```

```
1  /* File: stl-transform.h */
2  template <class Iterator1, class Iterator2, class Function>
3  Iterator2 transform(Iterator1 first, Iterator1 last,
4                      Iterator2 result, Function g)
5  {
6      for ( ; first != last; ++first, ++result )
7          *result = g(*first);
8
9      return result;
10 }
```

STL for_each( ) to Add using Function Object Add

```
1   #include <list>          /* File: fo-add.h */
2   #include <vector>
3   #include <algorithm>
4
5   class Add
6   {
7     private:
8       int data;
9     public:
10      Add(int i) : data(i) { }
11      int operator()(int value) { return value + data; }
12  };
13
14  class Print
15  {
16    private:
17      ostream& os;
18    public:
19      Print(ostream& s) : os(s) { }
20      void operator()(int value) { os << value << " "; }
21  };
```

STL for_each( ) to Add using Function Object Add ..

```
1   #include <iostream>      /* File: fo-add10.cpp */
2   using namespace std;
3   #include "fo-add.h"
4
5   int main()
6   {
7       list<int> x;
8       for (int j = 0; j < 5; ++j)    // Initialize x
9           x.push_back(j);
10
11      vector<int> y(x.size());
12      transform( x.begin(), x.end(), y.begin(), Add(10) );
13
14      for_each( y.begin(), y.end(), Print(cout) );
15      cout << endl;
16
17      return 0;
18  }
```

Other Algorithms in the STL

- min_element and max_element
- equal
- generate (Replace elements by applying a function object)
- remove, remove_if Remove elements
- reverse, rotate Rearrange sequence
- random_shuffle
- binary_search
- sort (using a function object to compare two elements)
- merge, unique
- set_union, set_intersection, set_difference