Object-Oriented Programming and Data Structures

COMP2012: Standard Template Library (STL) for Generic Programming

Cecia Chan
Brian Mak
Dimitris Papadopoulos

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:

S T L

iterators algorithms

containers

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

```
template <typename T>   /* File: arrayT.h */
class Array
{
  private:
    int _size;
    T* _value;

  public:
    Array<T>(int n = 10);    // Default and conversion constructor
    Array<T>(const Array<T>&); // Copy constructor
    ~Array<T>();

    int size() const { return _size; }
    void init(const T& k);

    Array<T>& operator=(const Array<T>& a);// Copy assignment operator
    T& operator[](int i) { return _value[i]; } // lvalue
    const T& operator[](int i) const { return _value[i]; } // rvalue
};
```

An Array Container Class Too

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

```
template <typename T>   /* File: array.h */
class Array
{
  private:
    T* _value;
    int _size;

  public:
    Array(int n = 10);   // Default and conversion constructor
    Array(const Array&); // Copy constructor
    ~Array();

    int size() const { return _size; }
    void init(const T& k);

    Array& operator=(const Array&);       // Copy assignment operator
    T& operator[](int i) { return _value[i]; } // lvalue
    const T& operator[](int i) const { return _value[i]; } // rvalue
};
```

Example: Use of Class Array

```
#include <iostream>      /* File: test-array.cpp */
using namespace std;
#include "array.h"
#include "array-constructors.h"
#include "array-op=.h"
#include "array-op-os.h"

int main()
{
    Array<int> a(3);
    a.init(98); cout << a << endl;
    a = a; a[2] = 17; cout << a << endl;

    Array<char> b(4);
    b.init('g'); b[0] = a[1]; cout << b << endl;

    const Array<char> c = b;
    // c[2] = 5; // Error: assignment of read-only location
    cout << c << endl;

    Array<int> d;
    d = a; cout << d << endl;
    return 0;
}
```

Constructors/Destructor of Class Array

```
template <typename T>   /* File: array-constructors.h */
Array<T>::Array(int n) : _value( new T [n] ), _size(n) { }

template <typename T> Array<T>::Array(const Array<T>& a)
    : Array(a._size)   // Delegating constructor
{
    for (int i = 0; i < _size; ++i)
        _value[i] = a._value[i];
}

template <typename T> Array<T>::~Array() { delete [] _value; }

template <typename T> void Array<T>::init(const T& k)
{
    for (int i = 0; i < _size; ++i)
        _value[i] = k;
}
```

Assignment Operators of Class Array: Deep/Shallow Copy

```
template <typename T>     /* File: array-op=.h */
Array<T>& Array<T>::operator=(const Array<T>& a) // Deep copy
{
    if (&a != this)       // Avoid self-assignment: e.g., a = a
    {
        delete [] _value;                // First remove the old data
        _size = a._size;
        _value = new T [_size];          // Re-allocate memory

        for (int j = 0; j <_size; ++j) // Copy the new data
            _value[j] = a[j];
    }

    return (*this);
}
```

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.

```
template <typename T>   /* File: array-op-os.h */
ostream& operator<<(ostream& os, const Array<T>& a)
{
    os << "#elements stored = " << a.size() << endl;

    for (int j = 0; j < a.size(); ++j)
        os << a[j] << endl;

    return os;
}
```

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.

```
template <typename T>   /* File: array-w-os-friend.h */
class Array
{
    template <typename S>
        friend ostream& operator<<(ostream& os, const Array<S>& x);

  private:
    T* _value;
    int _size;

  public:
    Array(int n = 10);   // Default and conversion constructor
    Array(const Array&); // Copy constructor
    ~Array();

    int size() const { return _size; }
    void init(const T& k);
    Array& operator=(const Array&);       // Copy assignment operator
    T& operator[](int i) { return _value[i]; } // lvalue
    const T& operator[](int i) const { return _value[i]; } // rvalue
};
```

Operator$\ll$ as a Friend Function Template ..

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

```
template <typename T>   /* File: array-op-os-friend.h */
ostream& operator<<(ostream& os, const Array<T>& a)
{
    os << "#elements stored = " << a._size << endl;

    for (int i = 0; i < a._size; ++i)
        os << a._value[i] << endl;

    return os;
}
```

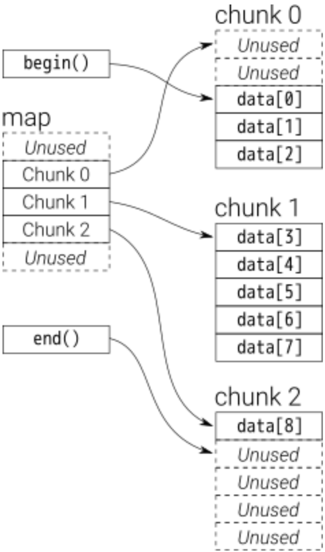
1. Sequence containers
   - Represent sequential data structures
   - Start from index/location 0
2. Associative containers
   - Non-sequential containers
   - Store (key, value) pairs
3. Container adaptors
   - adapted containers that support a limited set of container operations
4. "Near-containers" 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

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).
- Deque (double-ended queue)
  - Unlike STL vector, the elements of a deque are not stored contiguously; it uses a sequence of chunks of fixed-size arrays.
  - Like STL vector, the storage of a deque is automatically expanded/contracted as needed, but deque does not require copying of *all* the existing elements.
  - Allows fast insertion and deletion at both ends.

Deque (Double-Ended QUEue)



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>

```
template<typename T, typename Sequence = deque<T> >
class stack
{
  protected:
    Sequence c; // Underlying container

  public:
    typedef typename Sequence::value_type        value_type;
    typedef typename Sequence::reference         reference;
    typedef typename Sequence::const_reference   const_reference;
    typedef typename Sequence::size_type         size_type;

    // (Default) Constructor
    explicit stack(const Sequence& _c = Sequence()) : c(_c) { }
```

```
    // Return true if the stack is empty
    bool empty() const { return c.empty(); }

    // Return the number of elements in the stack
    size_type size() const { return c.size(); }

    // Return a R/W reference to the data at the first element
    reference top() { return c.back(); }

    // Read-only version of top()
    const_reference top() const { return c.back(); }

    // Create an element at the top of the stack and assign x to it
    void push(const value_type& x) { c.push_back(x); }

    // Shrink the stack by one. Note that no data is returned.
    void pop() { c.pop_back(); }
};
```

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

2	26		
2	13	...	0
2	6	...	1
2	3	...	0
2	1	...	1
	0	...	1

Example: Decimal to Binary Conversion

```
#include <iostream>      /* File: decimal2binary.cpp */
#include <stack>
using namespace std;

int main()  // Convert +ve decimal number to binary number using a stack
{
    stack<int> a;
    int x, number;

    while (cin >> number)
    {
        // Conversion: decimal to binary
        x = number;
        do { a.push(x % 2); x /= 2; } while (x > 0);

        // Print a binary that is stored on a stack
        cout << number << " (base 10) = ";
        while (!a.empty()) { cout << a.top(); a.pop(); }
        cout << " (base 2)" << endl;
    }

    return 0;
}
```

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

```
#include <iostream>     /* File: balanced-paren.cpp */
#include <stack>
using namespace std;

const char L_PAREN    = '('; const char R_PAREN    = ')';
const char L_BRACE    = '{'; const char R_BRACE    = '}';
const char L_BRACKET  = '['; const char R_BRACKET  = ']';
bool balanced_paren(const char* expr);

int main()  // To check if a string has balanced parentheses
{
    char expr[1024];
    cout << "Input an expression containing parentheses: ";
    cin >> expr;
    cout << boolalpha << balanced_paren(expr) << endl;
    return 0;
}

bool check_char_stack(stack<char>& a, char c)
{
    if (a.empty()) return false;
    if (a.top() != c) return false;
    a.pop(); return true;
}
```

Example: Balanced Parentheses ..

```
bool balanced_paren(const char* expr)
{
    stack<char> a;
    for (const char* s = expr; *s != '\0'; ++s)
        switch (*s)
        {
            case L_PAREN: case L_BRACE: case L_BRACKET:
                a.push(*s); break;

            case R_PAREN:
                if (!check_char_stack(a, L_PAREN)) return false;
                break;
            case R_BRACE:
                if (!check_char_stack(a, L_BRACE)) return false;
                break;
            case R_BRACKET:
                if (!check_char_stack(a, L_BRACKET)) return false;
                break;

            default: break;
        }

    return a.empty();
}
```

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

```
template<typename T, typename Sequence = deque<T> >
class queue
{
  protected:
    Sequence c; // Underlying container

  public:
    typedef typename Sequence::value_type        value_type;
    typedef typename Sequence::reference         reference;
    typedef typename Sequence::const_reference   const_reference;
    typedef typename Sequence::size_type         size_type;

    // (Default) Constructor
    explicit queue(const Sequence& _c = Sequence()) : c(_c) { }

    // Return true if the queue is empty
    bool empty() const { return c.empty(); }

    // Return the number of elements in the queue
    size_type size() const { return c.size(); }

    // Return a R/W reference to the data at the first element of the queue
    reference front() { return c.front(); }
```

Simplified STL Queue ..

```
    // Read-only version of front()
    const_reference front() const { return c.front(); }

    // Return a R/W reference to the data at the last element of the queue
    reference back() { return c.back(); }

    // Read-only version of back()
    const_reference back() const { return c.back(); }

    // Create an element at the end of the queue and assigns x to it
    // i.e., enqueue
    void push(const value_type& x) { c.push_back(x); }

    //  It shrinks the queue by one. Note that no data is returned.
    // i.e., dequeue
    void pop() { c.pop_front(); }
};
```

Example: Queue of int Data

```
#include <iostream>     /* File: int-queue-test.cpp */
#include <queue>
using namespace std;

void print_queue_info(const queue<int>& a) {
    cout << "\nNo. of data currently on the queue = " << a.size() << endl;
    if (!a.empty()) {
        cout << "First: " << a.front() << "\nLast: " << a.back() << endl; }
}
int main()
{
    queue<int> a; print_queue_info(a);
    a.push(4);    print_queue_info(a);
    a.push(15);   print_queue_info(a);
    a.push(26);   print_queue_info(a);
    a.push(37);   print_queue_info(a);
    a.pop();      print_queue_info(a);
    a.push(48);   print_queue_info(a);
    a.push(59);   print_queue_info(a);
    a.pop();      print_queue_info(a);
    a.pop();      print_queue_info(a);
    a.pop();      print_queue_info(a);
    a.pop();      print_queue_info(a);
    a.pop();      print_queue_info(a); return 0;
}
```

Part III

STL Iterators: Generalized Pointers

Pointers to Traverse an Array of a Basic Type

```
#include <iostream>      /* File: print-int-array.cpp */
using namespace std;

int main()
{
    const int LENGTH = 5;
    int x[LENGTH];

    for (int j = 0; j < LENGTH; ++j)
        x[j] = j;

    // x_end points to a non-existing element just beyond the array
    const int* x_end = &x[LENGTH];

    for (const int* p = x; p != x_end; ++p)
        cout << *p << endl;

    return 0;
}
```

- 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

```cpp
#include <iostream>         /* File: print-list.cpp */
using namespace std;
#include <list>             // STL list

int main()
{
    list<int> x;                    // An int STL list
    for (int j = 0; j < 5; ++j)
        x.push_back(j);             // Append items to an STL list

    list<int>::const_iterator p; // STL list iterator
    for (p = x.begin(); p != x.end(); ++p)
        cout << *p << endl;

    return 0;
}
```

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.

```
typedef int* Int_Iterator; /* File: find-int-iterator.cpp */

/* Actually this find function is already defined in STL */
Int_Iterator
find(Int_Iterator begin, Int_Iterator end, const int& value)
{
    while (begin != end && *begin != value)
        ++begin;

    return begin;
}
```

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

```
#include <iostream>      /* File: find-test.cpp */
using namespace std;
typedef int* Int_Iterator;

int main()
{
    const int SIZE = 10; int x[SIZE];
    for (int i = 0; i < SIZE; i++)
        x[i] = 2 * i;

    Int_Iterator begin = x; Int_Iterator end = &x[SIZE];
    while (true)
    {
        cout << "Enter number: "; int num; cin >> num;
        Int_Iterator position = find(begin, end, num);

        if (position == end)
            cout << "Not found\n";
        else if (++position != end)
            cout << "Found before the item " << *position << '\n';
        else
            cout << "Found as the last element\n";
    }
    return 0;
}
```

Why Are Iterators So Great?

```
template <class Iterator, class T> /* File: find-template.h */
Iterator find(Iterator begin, Iterator end, const T& value)
{
    while (begin != end && *begin != value)
        ++begin;

    return begin;
}
```

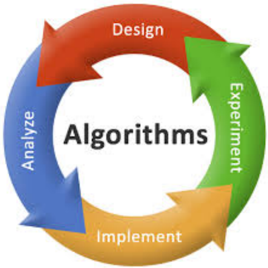
- Iterators allow us to separate algorithms from containers when they are used with templates.
- The new **find( )** function template contains no information about the implementation of the container, or how to move the iterator from one element to the next.
- The same **find**( ) function can be used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

Example: find( ) with a vector Iterator

```
#include <iostream>      /* File: find-iterator-test.cpp */
using namespace std;
#include <vector>

int main()
{
    const int SIZE = 10; vector<int> x(SIZE);
    for (int i = 0; i < x.size(); i++)
        x[i] = 2 * i;

    while (true)
    {
        cout << "Enter number: "; int num; cin >> num;
        vector<int>::iterator position = find(x.begin(), x.end(), num);

        if (position == x.end())
            cout << "Not found\n";
        else if (++position != x.end())
            cout << "Found before the item " << *position << '\n';
        else
            cout << "Found as the last element\n";
    }

    return 0;
}
```

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.

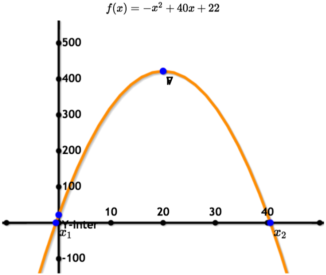
```
template <class Iterator, class T> /* File: stl-find.cpp */
Iterator find(Iterator first, Iterator last, const T& value)
{
    while (first != last && *first != value)
        ++first;

    return first;
}
```

Example: Using STL find( )

```
#include <iostream>      /* File: find-composer.cpp */
using namespace std;
#include <string>
#include <list>
#include <algorithm>

int main()
{
    list<string> composers;
    composers.push_back("Mozart");
    composers.push_back("Bach");
    composers.push_back("Chopin");
    list<string>::iterator p =
        find(composers.begin(), composers.end(), "Bach");

    if (p == composers.end())
        cout << "Not found." << endl;
    else if (++p != composers.end())
        cout << "Found before: " << *p << endl;
    else
        cout << "Found at the end of the list." << endl;

    return 0;
}
```

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.




```
/* File: init.h */
inline int quadratic(int x) { return -x*x + 40*x + 22; }

template <typename T>
void my_initialization(T& x, int num_items)
{
    for (int j = 0; j < num_items; ++j)
        x.push_back( quadratic(j) ); // Can you rewrite using lambda?
}
```

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

```
#include <iostream>      /* File: find-2nd-occurrence.cpp */
using namespace std;
#include <vector>
#include <algorithm>
#include "init.h"

int main()
{
    const int search_value = 341;
    vector<int> x;
    my_initialization(x, 100);

    vector<int>::iterator p = find(x.begin(), x.end(), search_value);

    if (p != x.end())    // Value found for the first time!
    {
        p = find(++p, x.end(), search_value); // Search again
        if (p != x.end())
            cout << search_value << " appears after " << *--p << endl;
    }
    return 0;
}
```

STL find_if( )

```
template <class Iterator, class Predicate> /* File: stl-find-if.cpp */
Iterator find_if(Iterator first, Iterator last, Predicate predicate)
{
    while (first != last && !predicate(*first))
        ++first;

    return first;
}
```

- 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

```
#include <iostream>      /* File: find-gt350.cpp */
using namespace std;
#include <vector>
#include <algorithm>
#include "init.h"

bool greater_than_350(int value) { return value > 350; }

int main()
{
    vector<int> x;
    my_initialization(x, 100);

    vector<int>::const_iterator p =
        find_if( x.begin(), x.end(), greater_than_350 );

    if (p != x.end())
        cout << "Found element: " << *p << endl;

    return 0;
}
```

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( )

```
#include <iostream>      /* File: fp-smaller-larger.cpp */
using namespace std;

int larger(int x, int y) { return (x > y) ? x : y; }
int smaller(int x, int y) { return (x > y) ? y : x; }

int main()
{
    int choice;
    cout << "Choice: (1 for larger; others for smaller): ";
    cin >> choice;

    int (*f)(int, int) = (choice == 1) ? larger : smaller;

    cout << f(3, 5) << endl;
    return 0;
}
```

Example: Array of Function Pointers — Calculator

```
#include <iostream>     /* File: fp-calculator.cpp */
using namespace std;
double add(double x, double y) { return x+y; }
double subtract(double x, double y) { return x-y; }
double multiply(double x, double y) { return x*y; }
double divide(double x, double y) { return x/y; } // No error checking

int main()
{
    double (*f[])(double x, double y) // Array of function pointers
        = { add, subtract, multiply, divide };

    int operation; double x, y;
    cout << "Enter 0:+, 1:-, 2:*, 3:/, then 2 numbers: ";
    while (cin >> operation >> x >> y)
    {
        if (operation >= 0 && operation <= 3)
            cout << f[operation](x, y) << endl; // Call + - * /
        cout << "Enter 0:+, 1:-, 2:*, 3:/, then 2 numbers: ";
    }
    return 0;
}
```

Example: Function Pointer as Lambda

```
#include <iostream>      /* File: fp-smaller-larger-lambda.cpp */
using namespace std;

int main()
{
    int choice;
    cout << "Choice: (1 for larger; others for smaller): ";
    cin >> choice;

    int (*f)(int, int);

    if (choice == 1)
        f = [] (int x, int y) { return (x > y) ? x : y; };
    else
        f = [] (int x, int y) { return (x > y) ? y : x; };

    cout << f(3, 5) << endl;
    return 0;
}
```

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

```
#include <iostream>      /* File: fo-greater-than.cpp */
using namespace std;
#include <algorithm>
#include <vector>
#include "init.h"
#include "fo-greater-than.h"

int main()
{
    vector<int> x; my_initialization(x, 100);
    int limit = 0;

    while (cin >> limit)
    {
        vector<int>::const_iterator p =
            find_if(x.begin(), x.end(), Greater_Than(limit)); // Call FO

        if (p != x.end())
            cout << "Element found: " << *p << endl;
        else
            cout << "Element not found!" << endl;
    }

    return 0;
}
```

STL find_if( ) with Function Object Greater_Than ..

```
class Greater_Than       /* File: fo-greater-than.h */
{
  private:
    int limit;
  public:
    Greater_Than(int a) : limit(a) { }
    bool operator()(int value) { return value > limit; }
};
```

- 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

```
#include <iostream>     /* File: fo-count.cpp */
using namespace std;
#include <vector>
#include <algorithm>
#include "fo-greater-than.h"

int main()
{
    vector<int> x;
    for (int j = -5; j < 5; ++j)
        x.push_back(j*10);

    // Count how many items are greater than 10
    cout << count_if(x.begin(), x.end(), Greater_Than(10)) << endl;

    return 0;
}
```

STL for_each( ) to Sum using Function Object

```
#include <iostream>      /* File: fo-sum.cpp */
using namespace std;
#include <list>
#include <algorithm>

class Sum
{
  private:
    int sum;
  public:
    Sum() : sum(0) { }
    void operator()(int value) { sum += value; }
    int result() const { return sum; }
};

int main()
{
    list<int> x;
    for (int j = 0; j < 5; ++j) x.push_back(j); // Initialize x
    Sum sum = for_each( x.begin(), x.end(), Sum() );
    cout << "Sum = " << sum.result() << endl; return 0;
}
```

```
/* File: stl-foreach.h */
template <class Iterator, class Function>
Function for_each(Iterator first, Iterator last, Function g)
{
    for ( ; first != last; ++first )
        g(*first);

    return g; // Returning the input function!
}

/* File: stl-transform.h */
template <class Iterator1, class Iterator2, class Function>
Iterator2 transform(Iterator1 first, Iterator1 last,
                    Iterator2 result, Function g)
{
    for ( ; first != last; ++first, ++result )
        *result = g(*first);

    return result;
}
```

STL for_each( ) to Add using Function Object Add

```
#include <list>          /* File: fo-add.h */
#include <vector>
#include <algorithm>

class Add
{
  private:
    int data;
  public:
    Add(int i) : data(i) { }
    int operator()(int value) { return value + data; }
};

class Print
{
  private:
    ostream& os;
  public:
    Print(ostream& s) : os(s) { }
    void operator()(int value) { os << value << " "; }
};
```

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

```
#include <iostream>      /* File: fo-add10.cpp */
using namespace std;
#include "fo-add.h"

int main()
{
    list<int> x;
    for (int j = 0; j < 5; ++j)   // Initialize x
        x.push_back(j);

    vector<int> y(x.size());
    transform( x.begin(), x.end(), y.begin(), Add(10) );

    for_each( y.begin(), y.end(), Print(cout) );
    cout << endl;

    return 0;
}
```

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