Object-Oriented Programming and Data Structures

COMP2012: Generic Programming — STL

Prof. Brian Mak
Prof. C. K. Tang

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

Generic Programming

- GP means programming with types as parameters.
- C++ supports GP through the template mechanism.
- Function templates allow you to create functions that work on different types of objects.
- Class templates allow you to create classes of different types of objects.
- Operator overloading further allows you to use the simpler syntax to operate objects of different types.
- Let's write a **Date** class and **Student** class, both of which supports the operator> function so that we may call **my_max( )** with **Date** and **Student** objects.

A Date Class That Overloads Operator>

```
const int days_in_month[ ] = {                                /* File: date.h */
    31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
class Date {                                          // Only for the year 2015
  private:
    int days;                                         // Must be within [1, 365]
  public:
    Date(int n): days((n < 1 || n > 365) ? 1 : n) { }
    bool operator>(const Date& x) const { return (days > x.days); }
    int month( ) const {
        for (int remain = days, m = 0; m < 12; ++m)
            if (remain <= days_in_month[m]) return m+1;
            else remain -= days_in_month[m];
        return -1;                            // Shouldn't reach this line of code
    }
    int day( ) const {
        for (int remain = days, m = 0; m < 12; ++m)
            if (remain <= days_in_month[m]) return remain;
            else remain -= days_in_month[m];
        return -1;                            // Shouldn't reach this line of code
    }
};
```

A Student Class That Overloads Operator>

```
class Student                                              /* File: student.h */
{
    friend ostream& operator<<(ostream& os, const Student& s)
    {
        os << "(" << s.name << " , " << s.dept << " , " << s.GPA << ")";
        return os;
    }

  private:
    string name;
    string dept;
    float GPA;

  public:
    Student(string n, string d, float x) : name(n), dept(d), GPA(x) { }

    bool operator>(const Student& s) const { return GPA > s.GPA; }
};
```

Example: Function Template + Operator Overloading

```
#include <iostream>                                    /* File: max-calls.cpp */
using namespace std;
#include "date.h"
#include "student.h"

template <typename T>
T my_max(const T& a, const T& b) { return (a > b) ? a : b; }

int main( ) {
    int x = 4, y = 8;
    cout << my_max(x, y) << " is a bigger number." << endl;

    string a("cheetah"), b("gorilla");
    cout << my_max(a, b) << " is stronger!" << endl;

    Date p(12), q(32); Date r = my_max(p, q);
    cout << "2015/" << r.month() << "/" << r.day() << " is later.\n";

    Student adam("Adam", "CSE", 3.8), joseph("Joseph", "MAE", 3.8);
    cout << my_max(joseph, adam) << " has a better GPA!" << endl;
}
```

Template + Operator Overloading: Be Careful

```
8 is a bigger number.
gorilla is stronger!
2015/2/1 is later.
(Adam , CSE , 3.8) has a better GPA!
```

- Read carefully the semantics of a function template before using it.
- **my_max( )** is originally designed to compare numerical values. If the 2 inputs are the same, it doesn't matter which one it returns.
- However, **Students** are objects! You use **my_max( )** to compare their GPAs which are just one component of the objects, but then return the whole object.
- Now, if their GPAs are the same, who is to returned?
- That is, the return type is not the same as the type of things you are comparing with.
- Otherwise, template + operator overloading + creativity may lead to powerful generic programming.

Part I

Introduction to STL

Example: Person and an Array Person_Container Class

```
class Person {                                          /* File: person.h */
  public:
    Person(string n, string a, string e)
        : name(n), address(a), email_address(e) { }
    string get_name( ) const;
    string get_address( ) const;
    string get_email_address( ) const;
  private:
    string name; string address; string email_address;
};
```

```
class Person_Container {                /* File: person-array-container.h */
  public:
    Person_Container(int n) : MAX_SIZE(n), size(0)
        { array = new Person [MAX_SIZE]; }
    int size( ) const { return size; }
    const Person& get_person(int i) const;
    void add_person(const Person& pers);
    void delete_person(int i);
  private:
    const int MAX_SIZE; int size;      // Number of Persons actually stored
    Person* array;                     // One-time pre-allocated storage
};
```

Container Class

Classes that maintain collections of objects are so common that they have been given a name: container classes.

- Let's write a program to maintain a collection of persons, and apply some operations on that collection.
- The operations on **Person_Container** can be:
  - member functions of the **Person_Container** class.
  - global functions that take a **Person_Container&** argument.
- Here we print mailing labels for all the persons, and send emails to invite them to our party.
- Note the similarities in both functions: they both set up a loop to do something for all persons in the container.
- We can expect that if we add more functions that do something with all persons, that these functions show the same similarities.

```
/* File: print-ml-array.cpp */
void print_mailing_labels(const Person_Container& pc)

    for (int i = 0; i < pc.size( ); ++i) {
        const Person& pers = pc.get_person(i);
        cout << pers.get_name( ) << endl;
        cout << pers.get_address( ) << endl;
        // ...
    }
}

/* File: invite-party-array.cpp */
void invite_to_party(const Person_Container& pc)
{
    for (int i = 0; i < pc.size( ); ++i) {
        const Person& p = pc.get_person(i);
        string command = "cat party.txt | mail ";
        command += p.get_email_address( );
        system( command.c_str( ) );                // Send invitation emails
    }
}
```

Array or Linked List?

- In some applications it is very convenient that we implement **Person_Container** with an array; the **get_person( )** member function takes only $O(1)$ (constant) time, and we use that member function a lot.
- However, in other applications we may find that we frequently need to merge two **Person_Container**'s into a single one, or split one **Person_Container** into two **Person_Container**'s.
- Now the fact that we use an array is a drawback (why?); a linked list would have been more practical in this case.
- So let's implement a container class called **Person_List** representing a list of **Persons**.

To Use a Linked List as a Container

The following interface functions are required:

- maintains a private pointer to the "current" element.
- **get_current( )** $\Rightarrow$ get the current element.
- **get_first( )** $\Rightarrow$ sets the pointer to the 1st item on the list.
- **get_next( )** $\Rightarrow$ sets the pointer to the next element.
- **get_prev( )** $\Rightarrow$ sets the pointer to the previous element.
- These functions return "-1" if there is nothing to point to.

We could, of course, add a member function **get_person(i)** that retrieves a person by index, but what would that do to the running time of **print_mailing_labels( )**?

Example: Operations on Person_List

```
/* File: print-ml-list.cpp */
void print_mailing_labels(const Person_List& pl) {
    if (pl.get_first( ) == -1)
        return;                                        // List is empty
    do {
        const Person& p = pl.get_current( );
        cout << p.get_name( ) << endl;
        cout << p.get_address( ) << endl;
    } while (pl.get_next( ) != -1);                    // End of list is reached
}

/* File: invite-party-list.cpp */
void invite_to_party(const Person_List& pl) {
    if (pl.get_first( ) == -1)
        return;                                        // List is empty
    do {
        const Person& p = pl.get_current();
        string command = "cat party.txt | mail ";
        command += p.get_email_address( );
        system( command.c_str( ) );                    // Send invitation email
    } while (pl.get_next( ) != -1);                    // End of list is reached
}
```

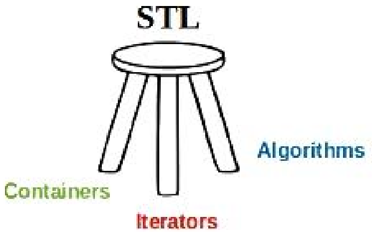
Similar Codes Again

- Suppose that we want to search for an item in a container.
- Conceptually, the search algorithm is independent of the type of element. However, in our examples, we would need separate functions for searching
  - a **Person** with a specific name in a **Person_Container**
  - a **Person** with a specific name in a **Person_List**
- Now, if we work on integers instead, we would need to implement a **Int_Container** first, then write separate functions for searching
  - a specific value in a **Int_Container**
  - a specific value in a **Int_List**

Three Concepts of Containers

- In the previous examples, we can distinguish 3 concepts about containers:
    - the kind of container (list-based, array-based)
    - the kind of objects stored in the container (**Person**, int)
    - the kind of operations on the elements stored in the container ("do something for each element").
- Containers are very common in programming, and several algorithms on container (searching for a specific element, sorting) occur in almost every non-trivial program.
- In our examples, there was a strong coupling between the three concepts.
- It is possible to remove (or strongly reduce) the strong coupling between containers, contained elements, and operations on the elements of the container by applying generic programming.

- 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 topics:



Part II

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 type in STL is better, so this is just for our understanding.

An Array Container Class

```
template <typename T>                                    /* File: arrayT.h */
class Array
{
  private:
    T* _value;
    int _size;
  public:
    Array<T>(int n = 10);      // Default and conversion constructor
    Array<T>(const Array& a);                     // Copy constructor
    ~Array<T>( );

    int size() const { return _size; }
    void init(const T& k);
    Array& operator=(const Array<T>& a);// 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& a);                     // Copy constructor
    ~Array( );

    int size() const { return _size; }
    void init(const T& k);
    Array& operator=(const Array& a);        // 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: array-test.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;
    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)
    : _value( new T[a._size] ), _size (a._size)
{
    for (int i = 0; i < _size; ++i) _value[i] = a._value[i];
}

template <typename T>
Array<T>::~Array( ) { delete [ ] _value; _value = 0; _size = 0; }

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

Assignment Operator of Class Array: Deep Copy

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

        _value = new T [_size];  // Re-allocate memory for new data
        _size = x.size( );

        for (int j=0; j < _size; ++j)            // Copy the new data
            _value[j] = x[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>& x)
{
    os << "#elements stored = " << x.size( ) << endl;

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

    return os;
}
```

Operator≪ as a Friend Function Template

- The Array class template may declare the operator≪ as a friend function inside the 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 or conversion constructor
    Array(const Array& a);                                    // Copy constructor
    ~Array( );
    int size() const { return _size; }
    void init(const T& k);
    Array& operator=(const Array& a);                    // Assignment operator
    T& operator[ ](int i) { return _value[i]; }   // lvalue
    const T& operator[ ](int i) const { return _value[i]; }   // rvalue
};
```

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.

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

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

    return os;
}
```

Containers in STL

1. Sequence containers
   - Represent linear data structures
   - Start from index/location 0
2. Associative containers
   - Non-sequential containers
   - Store key/value pairs
3. Container adapters
   - Implemented as constrained sequence containers
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 ..

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

- Containers of the same category share a set of similar, if not the same, public member functions (or public interface or algorithms).

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 middle O(1) at front/end	O(1) at any position
deque (doubly-ended queue)	O(1) random access	O(1) at front/back O(n) in 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 at a given position.
- erase(p): Remove an element.
- clear( ): Erase all elements.

Miscellaneous Operations:

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

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

Part III

STL Iterators

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 just beyond the array x
    const int* x_end = &x[LENGTH];

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

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`$\rightarrow$	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.
- STL sequence containers provide the begin( ) and end( ) to set an iterator to the beginning and end of a container.
- For each kind of container in the STL, there is an iterator type.
  - list<int>::iterator
  - vector<string>::iterator
  - deque<double>::iterator

Iterators to Traverse a Sequence Container ..

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

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

  typedef <a type expression> <type-synonym>

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

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];
    Int_Iterator begin = x; Int_Iterator end = &x[SIZE];
    for (int i = 0; i < SIZE; i++) x[i] = 2 * i;

    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";
    }
}   /* Compile as: g++ find-int.cpp find-test.cpp */
```

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 an Iterator

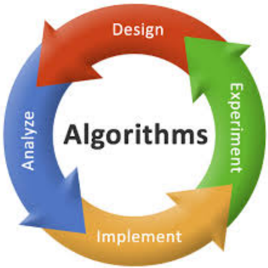
```
#include <iostream>                    /* File: find-iterator-test.cpp */
using namespace std;
#include <vector>
int main( )
{
    const int SIZE = 10; vector<int> x(SIZE);
    for (int i = 0; i < x.size( ); i++) x[i] = 2 * i;

    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";
    }
}
```

Part IV

STL Algorithms



STL Algorithm

- The STL not only contains 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 linearly 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;
}
```

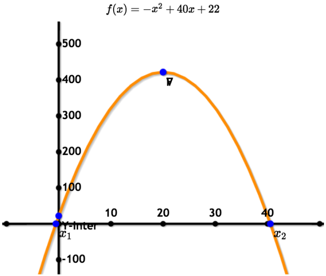
Example: Using STL find( )

```
#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");
    composers.push_back("Beethoven");
    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;
}
```

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) );
}
```

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

```
#include <iostream>                        /* File: find-2nd-occur.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;
    }
}
```

STL find_if( )

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

- find_if( ) is a more general algorithm than find( ) in that it stops when a condition is satisfied.
- This allows partial match, or match by keys.
- The condition is called a predicate and is implemented by a boolean function.
- 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>::iterator p =
        find_if( x.begin(), x.end(), greater_than_350 );

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

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.
- If you "man 3 qsort" on a Linux terminal, you will see:

```
void qsort(void *base, size_t nmemb, size_t size,
    int (*compare)(const void *, const void *))
```

- The 4th argument, compare here, is a function pointer, whose prototype is:

```
int (*)(const void*, const void*);
```

- Similarly, the type for the template `max( )` function pointer we talked before is:

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

Function Pointer Example: min( ) and max( )

```
#include <iostream>                                /* File: fp-min-max.cpp */

int max(int x, int y) { return (x > y) ? x : y; }
int min(int x, int y) { return (x > y) ? y : x; }

int main( )
{
    int choice;
    std::cout << "Choice:  (1 for max; others for min:  ";
    std::cin >> choice;

    int (*f)(int x, int y);
    f = (choice == 1) ? max : min;

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

Function Pointer Example: 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( )
{   // Array of function pointers
    double (*f[ ])(double x, double y) = { add, subtract, multiply, divide };

    int operation; double x, y;
    cout << "Enter 1:+, 2:-, 3:*, 4:/, then 2 numbers:  ";

    while (cin >> operation >> x >> y)
    {
        if (operation > 0 && operation < 5)
            cout << f[--operation](x, y) << endl;          // Call + - * /
        cout << "Enter 1:+, 2:-, 3:*, 4:/, then 2 numbers:  ";
    }
}
```

Function Pointer Example: Sorting by qsort( )

```
#include <iostream>                            /* File: fp-qsort.cpp */
using namespace std;
#include <cstdlib>               // Contains the qsort function declaration

int i_compare(const void* i, const void* j) // Prototype required by qsort
{
    return *(static_cast<const int*>(i))          // Casting is needed
           - *(static_cast<const int*>(j));
}

int main( )
{
    int data[ ] = { 3, 7, 5, 1, 9 };
    int num_data = sizeof(data)/sizeof(data[0]);
    qsort(data, num_data, sizeof(data[0]), i_compare); // Quicksort on data

    for (int j = 0; j < num_data; ++j)
        cout << data[j] << ' ';
    cout << 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 pointer is just one example of function objects.
- An object can be called if it supports the operator( ).
- A function object must have at least the operator( ) overloaded, and 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 */
#include <algorithm>
#include <vector>
#include "init.h"
#include "fo-greater-than.h"

int main( )
{
    std::vector<int> x; my_initialization(x, 100);
    int value = 0;

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

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

```
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 function object g
Greater_Than g(350);
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])
```

```
#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;
}
```

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;
}
```

STL Algorithms: for_each( ) and transform( )

```
/* 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:
    std::ostream& os;
  public:
    Print(std::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)
        x.push_back(j);                                         // Initialize x

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

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

STL Stream Iterators

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

int main( ) {
    list<int> x; vector<int> y;

    // An istream iterator only accepts int
    istream_iterator<int> input_iterator(cin);

    for (int j = 0; j < 5; ++j, ++input_iterator)
        x.push_back(*input_iterator);     // Initialize with iterator's content

    // Copy x to y after adding 77 to x's items
    // back_insert(y) calls y's push_back( )
    transform(x.begin( ), x.end( ), back_inserter(y), Add(77) );

    // Print to an ostream iterator linked to cout with newline separator
    copy(y.begin( ), y.end( ), ostream_iterator<int>(cout, "\n"));
}
```

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

Final Example: Use of STL

```
#include <iostream>                                    /* File: stl-example.cpp */
using namespace std;
#include <vector>
#include <string>
#include <iterator>
int main( ) {
    vector<string> people; string name;
    while (cin >> name) people.push_back(name);

    cout << "With duplicates:" << endl;
    copy(people.begin( ), people.end( ),
         ostream_iterator<string>(cout, "\n"));

    vector<string> friends;
    copy( people.begin( ), people.end( ), back_inserter(friends) );
    sort( friends.begin( ), friends.end( ) );

    cout << endl << "Without duplicates:" << endl;
    unique_copy( friends.begin( ), friends.end( ),
                 ostream_iterator<string>(cout, "\n") );
}
```

Final Example: Use of STL ..

Input:

```
Brian Rene Gary Anna Brian Conny Raymond Raymond Brian
```

Output:

```
With duplicates:
Brian
Rene
Gary
Anna
Brian
Conny
Raymond
Raymond
Brian

Without duplicates:
Anna
Brian
Conny
Gary
Raymond
Rene
```