COMP 2012H Honors Object-Oriented Programming and Data Structures

Topic 14: Standard Template Library

Dr. Desmond Tsoi

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

Part I

Generic Programming Again

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 the generic operator function syntax to work for objects of user-defined new 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 non-leap years
{
  private:
    int days;  // Days in a non-leap year; 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 = days, m = 0; m < 12; ++m)
            if (remain_days <= days_in_month[m]) return m+1;
            else remain_days -= days_in_month[m];
    }

    int day() const
    {
        for (int remain_days = days, m = 0; m < 12; ++m)
            if (remain_days <= days_in_month[m]) return remain_days;
            else remain_days -= days_in_month[m];
    }
};
```

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

```cpp
#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 date1(120), date2(300); Date r = my_max(date1, date2);
    cout << r.month() << "/" << r.day() << " is a later date.\n";

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

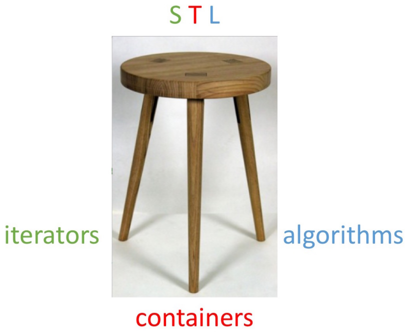
Template + Operator Overloading: Be Careful

```
8 is a bigger number.
gorilla is stronger!
10/27 is a later date.
(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 return?
- 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.

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 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 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:
    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; }

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

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

        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 ..

Type of Container	STL Containers
Sequence	vector, list, deque
Associative	map, multimap, multiset, set
Adapters	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.

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

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

Part III

STL Iterators: Generalized Pointers

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 STL sequence container, 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;
}
```

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;
}
```

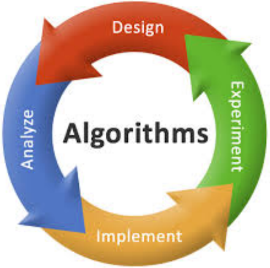
- Iterators allow us to separate algorithms from containers when they are used with templates.
- The new **find()** function template contains no information about the implementation of the container, or how to move the iterator from one element to the next.
- The same **find**() function can be used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

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;
}
```

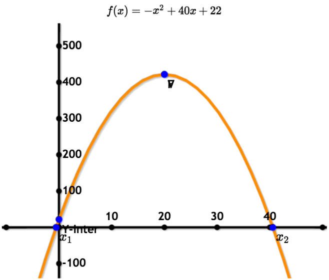
Example: Using STL find()

```
#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;
}
```

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.




```
/* 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-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;
    }
}
```

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>::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 type is:

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

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

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

Function Pointer Example: min() and max()

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

int my_max(int x, int y) { return (x > y) ? x : y; }
int my_min(int x, int y) { return (x > y) ? y : x; }

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

    int (*f)(int x, int y);
    f  = (choice == 1) ? my_max : my_min;

    cout << f(3, 5) << 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()
{
    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: ";
    }
}
```

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 */
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>::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;
    }
}
```

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 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])*

Other Algorithms in the STL

- count_if
- for_each
- transform
- 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

That's all!

Any questions?