Topic 4

STL Basics

The Standard Template Library

資料結構與程式設計
Data Structure and Programming

Sep, 2012

In this class,

we will introduce several **special** types of

Data Structures,

for example, list, array, set, map, hash, graph, etc.

Some people call them

Abstract Data Types (ADT)

or **(an easier-to-understand name)**

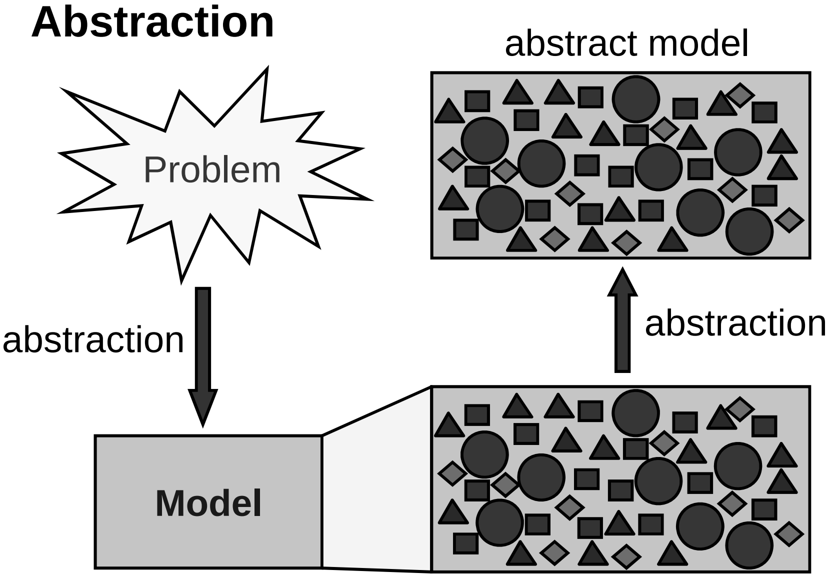
Container Classes

Pablo Picasso, “Accordionist”

Saver Containers

Abstract ?? Containers??

Abstraction



Data Types

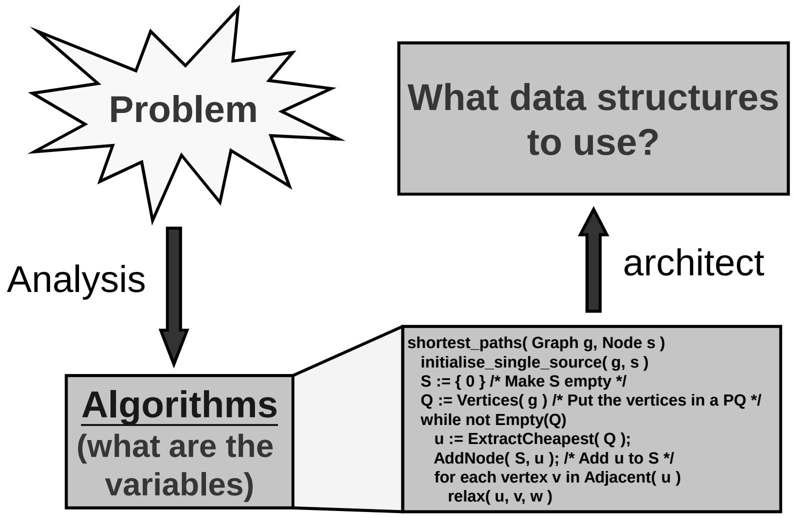
- "A data type, as defined in many object-oriented languages, is a class"
  1. Data member
     - Define data
  2. Member functions
     - Define operations

So, what does the "Abstract" in "Abstract Data Type" mean?

Some Quotes about ADT...

- "...precisely specified independent of any particular implementation"
- "You don't know how the ADT computes, but you know **what** it computes"
- "The implementer of the class can change the implementation for maintenance, bug fixes or optimization reasons, without disturbing the client code"

ADT in Programming (1)



ADT in Programming (2)

- When you write an OOP, you may start from thinking about the algorithms and data structure, and you may find that you need some “special classes” for ---
  1. Storing objects in a queue
  2. Enumerating objects you have stored
  3. Sorting objects
  4. Performing union or intersection on set of objects
  5. Associating one set of objects with another
  6. Finding some objects after they are stored
  7. Conceptually connecting objects as a graph

  etc...

ADT in Programming (3)

- Obviously, these kinds of classes are not specific to any type of algorithms
  - In other words, they can be implemented independently of the algorithms that use them
- What they provide ---
  - Interface functions to operate on the data stored in the class
  - The implied complexity of these functions
- What they don't show (Abstracted away...) ---
  - What are the data members inside?
  - How the functions are implemented?

ADT in Programming (4)

- That's why they are called "Abstract Data Types", or "Container Classes", and usually treated as special "utilities" for a programmer
  - Examples are:
    - List, array, queue, stack, set, map, heap, hash, string, bit vector, matrix, tree, graph, etc.
- The more and cleverer you use them, the better your program will be
  - That's the main purpose of learning this course

Classification of ADTs

1. Linear (Sequence) Data Types
   - List, array, queue, stack
2. Associative Data Types
   - Set, map, hash, heap
3. Topological Data Types
   - Tree, graph
4. Miscellaneous Types
   - String, bit vector, matrix

◆ Usually OOP programmer will implement these classes just once *(or adopt the existing ones)*, and later utilize them in various programs

The rationale for how these libs are implemented, and why, will be described in later topics. Here we will first learn how to use them wisely….

The one we are going to use is called "Standard Template Library (STL)"

(A good STL reference: http://www.sgi.com/tech/stl/)

The Standard Template Library

- Why template libraries?
- Why standard?
- The standard template libraries
  1. Container classes
  2. Iterators
  3. Algorithms
  4. Functional object
  5. Utility
  6. Memory allocation

Template Class

- When the methods of a class can be applied to various data types
  - Specify once, apply to all
  - Container classes

  e.g.

```
template <class T>
class vector
{
    ....
};
--------------------
vector<int>             arr;
vector<vector<int> >    arr2D;
```

➔ [note] make sure a space between “> >”

Template's Arguments

- Can also contain expression
  - However, the 1st argument must be class name

e.g.

```
template<class T, int SIZE>
class Buffer
{
   T   _data[SIZE];
};
------------------------------
Buffer<unsigned, 100>  uBuf;
Buffer<MyClass, 1024>  myBuf;
```


Template Function

- A common method/algorithm that can be applied to various data types

e.g.

```
template <class RandomAccessIterator,
          class StrictWeakOrdering>
void sort(RandomAccessIterator first,
          RandomAccessIterator last,
          StrictWeakOrdering comp);
---------------------
int arr[100] = { …. };
sort(arr, arr+100, less<int>());
```

Template Function's Arguments

```
template<class T> T* test() { … }
template<class T> void test2(T* a) { … }
template<class T> T* test3(T* a) { … }
-------
int main()
{
   int* a = …
   test1<int>();     // or test1();  ??
   test2<int>(a);    // or test2(a); ??
   test3<int>(a);    // or test3(a); ??
   …
}
```


Template Functional Object

```
template <class A1, class A2, class R>
   binary_function;

template <class T>
   struct less
   : binary_function<T, T, bool>;
```

- sort(arr, arr+100, less<int>());
- template <class A, class B, class C>
  class MyClass : public binary_function<A, B, C>

Template Class/Function vs. Function Overload vs. Functional Object

To maximize the code reuse (less code)

- Template
  - Class template
    - Same storage method, different data types
  - Function template
    - Same algorithm flow, different data types
- Function overloading
  - Same function name, different function arguments
- Functional object
  - Same algorithm flow, different functional methods as "arguments"

Standard Template Library (STL)

- First drafted by Alexander Stepanov and Meng Lee of HP in 1992
  - Became IEEE standard in 1994
- A C++ library of container classes, algorithms, and iterators
  - Provides many of the basic algorithms and data structures of computer science
- The STL is a *generic* library
  - Platform independent
  - Its components are heavily parameterized
  - Almost every component in the STL is a template.
- An useful reference: http://www.sgi.com/tech/stl/

With STL, people can…

- ◆ Develop software without the need to implement in-house container classes
- ◆ Create prototype more quickly
- ◆ Share the programs with other easily
- ◆ Port to different machine/OS platforms

[Note] However, in order to make STL generic, people sacrifice some performance and robustness

➔ Many commercial tools choose to implement their own TL's.

Standard Template Library

1. Container classes
   - list, slist, vector, deque, map, set, multimap, multiset, etc
   - Adaptors: stack, queue, etc
   - Non-template: string, bit_vector, etc
2. Iterators
   - Trivial iterator, input iterator, output iterator, forward iterator, bidirectional iterator, random access iterator, etc
3. Algorithms
   - for_each, sort, partial_sum, sort, find, copy, swap, etc.
4. Functional object
   - plus, minus, less, greater, logical_and, etc
5. Utility
   - Rational operators, pair, etc
6. Memory allocation
   - alloc, pthread_alloc, construct, uninialized_copy, etc

STL Container Classes

list<T>	doubly-linked list
slist<T>	singly-linked list
vector<T>	dynamic array
deque<T>	vector + O(1) delete/remove front
map<K, V>	map K to V; 1 to 1
multimap<K, V>	map K to V; many to 1
set<T>	set of T type elements; no repeat
multiset<T>	set of T type elements; allow repeat

Iterators in STL

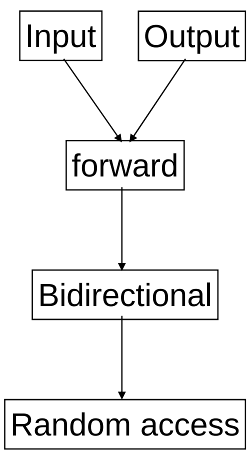
- To manipulate data in STL, you need to know iterator first!
- As its name suggests, iterator is to traverse data in a container class
  - list<int>::iterator li;
    for (li = myList.begin(); li != myList.end(); li++)
    { … }
- The STL defines several different concepts related to iterators, several predefined iterators, and a collection of types and functions for manipulating iterators.
  - Different algorithms/functions may take different types iterators

Operator Overload in Iterators

◆ Let “li” be an iterator

*li	dereference to access the object
li++	point to the next object in a range
li--	point to the previous object in a range
li + n	point to the next n object in a range
li - n	point to the previous n object in a range

Hierarchy of Iterators



Container	Iterator type	Operators
list	bidirectional	*, ++, --
slist	forward	*, ++
vector	random access	*, ++, --, +/-
deque	random access	*, ++, --, +/-
map	bidirectional	*, ++, --
multimap	bidirectional	*, ++, --
set	bidirectional	*, ++, --
multiset	bidirectional	*, ++, --
Adaptors	none	N/A

Iterators

- A generalization of pointers
  - They are objects that point to (data) objects in a class
  - Often used to iterate over a range of objects

  ➔ If an iterator points to one element in a range, then it is possible to increment it so that it points to the next element

  e.g.
  ```
  list<int>::iterator li;
  for (li = myList.begin(); li != myList.end(); li++) { … }
  ```
- Think: if "it" is currently pointing to a data in a container class, how does it move to the next data?

➔ Don't need to reveal the internal implementation of the container class
  - e.g. How does the data in "list<T>" get connected?
    - Another class listNode<T>? Pointers?
    - By "wrapping" the list nodes with iterators and providing interface functions (operators) like *, ++, --, we can traverse the list<T> without knowing how list<T> is implemented

Iterators

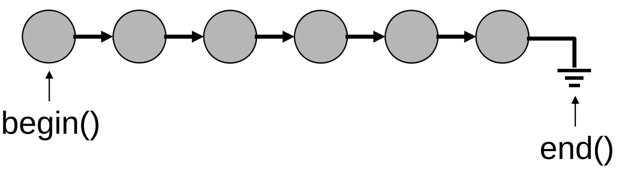
- An interface between containers and algorithms
  - Algorithms typically take iterators as arguments, so a container needs only provide a way to access its elements using iterators.
  - e.g.
  ```
  void sort(RandomAccessIterator first,
            RandomAccessIterator last,
            StrictWeakOrdering comp);
  ```
  ➔ Note: `RandomAccessIterator` does not tie to any container classes
- Make your code container class independent
  - ```
    typedef vector<int> MyContainer;
    ...
    MyContainer myData;
    for (MyContainer::iterator
         it = myData.begin();
         it != myData.end(); it++) { .... }
    ```
    ```

What does list<T>::iterator mean?

- list<T>::iterator li = L.begin();
- “iterator” is a class defined in list<T>? (Inner class? Or typedef?)
  - How can we have a same class name for different container classes?
    - e.g. list<T>::iterator, vector<T>::iterator...
  - How can we define a class that can access the internal implementation of the container classes, and provide interfaces for users’ operations?

➔ Covered in later homework

“end()” is actually “pass-the-end”



- end() points to the next to the last element
- Why not the last element?
- Think:
  - for (li = L.begin(); li != L.end(); li++)...
    - What’s the problem?
  - for (li = L.begin(); li < L.end(); li++)...
    - What’s the problem?

iterator vs. const_iterator

- Think:
  (*li) return the reference to the data
  - *li = 10 ← Can be used in LHS (writeable)
- What if the container is a "const"?
  - const list<int> ll;
    ...
    list<int> iterator li = ll.begin();
    *li = 10; ← Compile error!! (Why?)
- "const_iterator" is to ensure (*li) return const reference to the data.
  - const container MUST use const_iterator

Example #1

```
int arr[4] = { 3, 8, 4, 9 };

int main()
{
   sort(arr, arr + 4, less<int>());
   for (int i = 0; i < 4; i++)
      cout << arr[i] << endl;
   return 0;
}
```

- Note:
  ```
  void sort(RandomAccessIterator first,
            RandomAccessIterator last,
            StrictWeakOrdering comp);
  ```
- Use "int*" as trivial iterators

Example #2

```
list<char> arr;
vector<int> brr;
…..
list<char>::iterator li;
for (li = arr.begin(); li != arr.end(); li++)
{ … }

vector<int>::iterator vi;
for (vi = brr.begin(); vi != brr.end(); vi++)
{ … *vi = 8; … }
for (size_t i = 0, n = brr.size(); i < n; ++i)
{ … brr[5] = 8; … }
// need to make sure brr.size() doesn't change
```


Example #3

```
list<A*> ll;
ll.push_back(new A(3));
ll.push_front(new A(5));
…
while (!ll.empty()) { // cf. ".size()"
   A* a = ll.front();
   ll.pop_front();
   ...
   delete a;  // explicitly delete 'a'
}
```

Example #4

```
◆ vector<int> arr(10);
  // Initial size = 10
  for (int i = 0; i < 10; ++i) arr[i] = i;
◆ vector<int> arr;
  arr[0] = 10;     // crash!!! arr is an empty array
◆ vector<int> arr;
  arr.reserve(10);
  // Initial capacity = 10, size = 0
  arr[0] = 10;   // no crash; but not good...
  // size is still 0 ➔ Try: cout << arr.size() << endl;
◆ vector<int> arr;
  for (int i = 0; i < 10; ++i) arr.push_back(i);
◆ vector<int> arr; arr.resize(10); // what's the difference?
  for (int i = 0; i < 10; ++i) arr[i] = i;
◆ vector<int> arr; arr.reserve(10);// what's the difference?
   for (int i = 0; i < 10; ++i) arr.push_back(i);
◆ vector<int> arr(5);
  // Initial size = 5
  arr.resize(10);
```


Example #5

```
◆ map<string, int> scores;
  scores["Mary"] = 100;
  scores["John"] = 97;
  cout << scores["Mary"] << endl;
◆ map<const char*, int> scores;
  scores["Mary"] = 100;
  scores["John"] = 97;
  cout << score["Mary"] << endl;
  // Not good; may get garbage; why?
◆ map<string, int> scores;
  scores["Mary"] = 100;
  // What's scores.size()?
  cout << scores["John"] << endl;
  // What will you see? Crash?
  // What's scores.size()?
```

Example #6

- If we want to know if someone is already in the map...
  - ```
    map<string, int> scores;
    map<string, int>::iterator
          mi = scores.find("John") ;
    if (mi != scores.end())  // found!!
       cout << (*mi).first << " = "
            << (*mi).second << endl;
    ```
  - ➔ **`(*mi): pair<string, int>`**
  - ➔ **`Don't do (*mi) if NOT found!!`**
- **`If we wants to insert something into the map, and wants to know if we succeed...`**
  - ```
    map<string, int> scores;
    pair<map<string, int>::iterator, bool> p
    = scores.insert(make_pair("John", 100));
    ```
  - // If succeeds, p.first = iterator to the newly inserted,
  - //   and p.second = true;
  - // If fails, p.first = iterator to the existing object, ← no overwrite
  - //   and p.second = false;

Adaptor Classes

- Use "adaptor class" to implement containers on top of other ADTs
  - For example,
    ```
    template <class T, class C = Array<T> >
    class Stack {
       C    _elements;
    public:
       // only define operations
       // that make sense to "stack"
       // e.g. push(), pop(), top(), etc
    };
    ```
- STL adaptor classes
  - stack<class Value, class Container = deque>
  - queue, priority_queue

What is “pair”

- Template:

```
template <class First, class Second>
struct pair
{
   First   first;
   Second  second;
};
```

- The following are the same

```
scores.insert(make_pair("John", 100));
= scores.insert
  (pair<string, int>("John", 100));
= scores.insert
  (map<string, int>::value_type("John", 100));
```


Be careful when using map’s [ ] insertion...

- Note that when we do:

```
scores["Mary"] = 100;     or
cout << scores["John"] << endl;
```

we may insert a new element to the map if there is no element with the key value.

  - No matter LHS or RHS
  - Don’t use it to test the existence of an element
  - This operator [] cannot be a const member function
    ➔ Can not be called if map is a const
- Actually, m[k] is equivalent to
  (*((m.insert(value_type(k, data_type()))).first)).second
  ➔ Strictly speaking, this member function is unnecessary: it exists only for convenience.