Topic 10

Software Engineering

資料結構與程式設計

Data Structure and Programming

11/11/2015

Why Software Engineering?

- **Software development is hard** !
- **Important to distinguish "easy" systems** (*one developer, one user, experimental use only*) **from "hard" systems** (*multiple developers, multiple users, products*)
- **Experience with "easy" systems is misleading**
  - *One person techniques do not scale up*
- **Analogy with bridge building:**
  - Over a stream = easy, one person job
  - Over River Severn … ? (*the techniques do not scale)*

Source: http://www.csc.liv.ac.uk/~igor/COMP201/

Why Software Engineering ?

- The problem is ***complexity***
- Many sources, but ***size*** is key:
  - UNIX contains 4 million lines of code
  - Windows 2000 contains 10^8 lines of code

Software engineering is about managing this complexity.

Source: http://www.csc.liv.ac.uk/~igor/COMP201/

What is software engineering?

Software engineering is an engineering discipline which is concerned with all aspects of software production

Software engineers should

- Adopt a systematic and organized approach to their work
- Use appropriate tools and techniques depending on
  - the problem to be solved,
  - the development constraints and
  - the resources available

Source: http://www.csc.liv.ac.uk/~igor/COMP201/

What is a software process?

- A **set of activities** whose goal is the development or evolution of software
- Generic activities in all software processes are:
  - **Specification** - what the system should do and its development constraints
  - **Development** - production of the software system
  - **Validation** - checking that the software is what the customer wants
  - **Evolution** - changing the software in response to changing demands

Source: http://www.csc.liv.ac.uk/~igor/COMP201/

What are the attributes of good software?

The software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable

- **Maintainability**
  - Software must evolve to meet changing needs
- **Dependability**
  - Software must be trustworthy
- **Efficiency**
  - Software should not make wasteful use of system resources
- **Usability**
  - Software must be usable by the users for which it was designed

Source: http://www.csc.liv.ac.uk/~igor/COMP201/

Software Maintainability

- Facts
  - Source code size will grow
  - Multiple people are involved
  - Spec may change; bugs may occur
- Think:
  - Code size growth should not lead to a mess
  - One person's work should not hinder others from making progress
  - Incremental change vs. entire code rewrite
- What should you do?

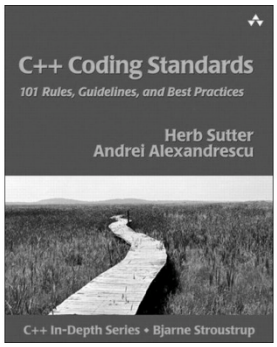
Software Maintainability

1. NO duplicated codes
   - Usually resulted from copy-and-paste
   - Create functions for the common parts
2. NO long function code
   - Divide it into multiple functions, or
   - Extract some common or frequenctly-used parts as sub-functions

   ➔ Keep It Simple and Short (KISS principle)
3. Good and consistent coding style
   - Especially naming convention
   - Source code layout
   - The best comment is no comment (self-documented)

C++ Coding Guidelines

- The first step in exercising software engineering principle is to follow the coding guidelines in the software development process
- This is an art.
  No universally correct answer.
- Google “C++ coding guideline”...



Coding Style --- Naming (FYI)

- Variable names
  - numStudents, isDone...
- Class names
  - LuxuryCar, BinaraySearchTree,...
- Function names
  - checkNumber(), computeScore()...
- #define / enum constant
  - RANGE, MAX_COLORS,...
- Class data members
  - _name, _id, _score,...
- Static, global variables / functions (optional)
  - nameMap_g, count_s, _memMgr_s, checkSum_s()...

Coding Style --- Look and Feel (FYI)

1. Proper indentation
   - 3 (or 2, or 4) spaces right for the codes within a new scope
   - Do not use "tab" ➔ platform dependent ➔ Use "space"
   - Try to turn off "auto indentation"
2. Proper alignment
   - ```
     if ((numStudents >= 30) &&
         (numStudents <= 80))
     ```
   - ```
     cout << "Hello, today is " << printDate()
          << ". Welcome to my world!!" << endl;
     ```
3. Braces { }
   - ```
     void function()
     {
     }
     ```
   - ```
     if (boolExpression) {
     }
     ```


Software Dependability

- Facts
  - Where there is a software, there is a bug.
  - The only way to enhance software dependability is ➔ **Test, and more tests**.
- Think:
  - What is an "experienced" coder?
  - Experience in:
    - "Spontaneous coding" (but with GOOD coding styles)
    - Debugging (to find the bug and to fix the bug)
  - You must get yourself familiar with debugger!!!
- The ultimate goal
  - When you see the bug, you know the possible cause(s)
  - No overnight (over-the-meal) bug

Software Dependability

- Regression test
  - A systematic mechanism to
    1. Collect and organize testcases
    2. Routinely run the testcases
    3. Make sure the newly added codes can still pass the testcases
    4. Check in new testcases for newly added codes
- Source code version control
  - A tool/database to centralize different versions of source codes
  - Differences between different versions are recorded incrementally, with logs and histories for later reference

Software Efficiency

- Facts
  - 80-20 rule
    - 80% (or more) resources (run time / memory) are consumed or controlled by 20% (or less) of the codes
  - Don't be picky about the efficiency of the 80% less critical codes
    - Higher priorities: maintainability, dependability
    - ➔ Even though they may have negative effects on the efficiency
    - ➔ However, negligence on the maintainability and dependability may lead to unstructured codes and eventually jeopardize the efficiency
- What you should do?
  - Equip basic instincts about the implied complexity of the data structure and algorithm
  - Know when to be picky and when to let go

Software Usability

- Facts
  - Usability factors = usage flow, user interface, ease of use, usage consistency
  - 80-20 rule
    - 80% (or more) of the code is not related to user friendliness
    - However, 20% (or less) of the code determines how your program is appreciated by others
- Importance of the minority
  - Decisions about the above usability factors determine the architecture of the code/framework
  - Later change is hard

➔ Plan at first!!

Discipline & Practice.

Take advantage of right tools and libraries

Make good use of proper data structures

Remember,

we have introduced several special types of Data Structures,

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

People call them

Abstract Data Types (ADT)

or

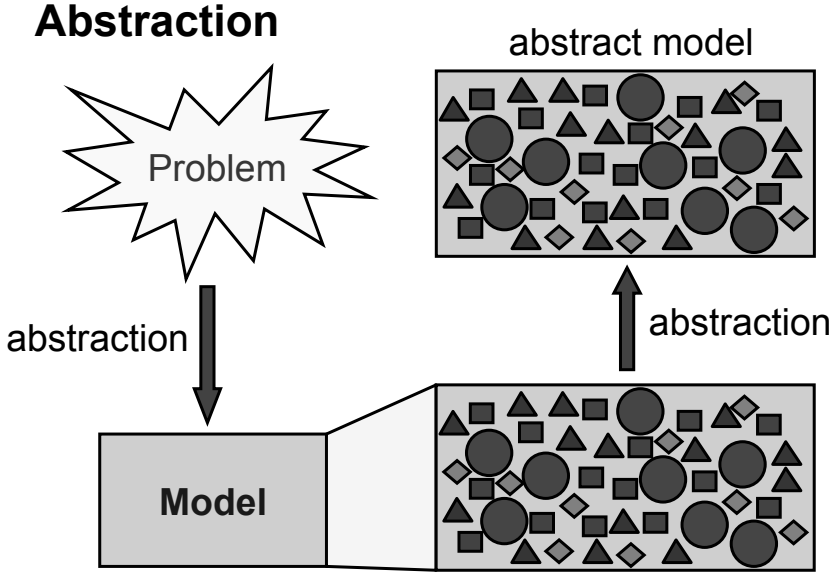
Container Classes

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...

- "... 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

- 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

- 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

Recall: 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

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

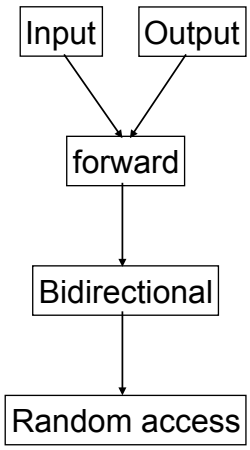
- To manipulate data in STL, you need to know iterator first!
- As its name suggests, iterator is to traverse data in a container class
  - list<int>::iterator li;
    for (li = myList.begin(); li != myList.end(); li++)
    { … }
- The STL defines several different concepts related to iterators, several predefined iterators, and a collection of types and functions for manipulating iterators.
  - Different algorithms/functions may take different types of 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 (1/2)

- 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 (2/2)

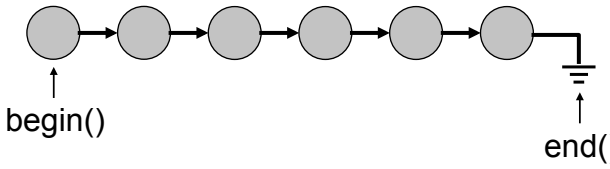
- An interface between containers and algorithms
  - Algorithms typically take iterators as arguments, so a container needs only provide a way to access its elements using iterators.
  - e.g.

```
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++) { .... }
```


"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

STL 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

STL 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
```


STL 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'
}
```

STL 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);`

STL 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 << scores["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;

What is "pair"

- ```
  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.

Conclusion

- Learning software engineering
  - Maintainability, dependability, efficiency, usability.
- Abstract data structure (ADT)
  - Implementation vs. user interface
- Standard Template Library (STL)