CS32 Discussion
Week 6

Muhao Chen
muhaochen@ucla.edu
http://yellowstone.cs.ucla.edu/~muhao/
Outline

• Recursion
• Template Classes
• STL Containers
Recursion

• Function-writing technique where the function refers to itself.

• Recall the following function:

```c
int factorial(int n)
{
    if (n <= 1)
        return 1;
    return n * factorial(n - 1);
}
```

• Let us talk about how to come up with such a function.
Pattern

• How to Write a Recursive Function for Dummies
  1. Find the base case(s).
     • What are the trivial cases? e.g. empty string, empty array, etc.
     • When should the recursion stop?
  2. Decompose the problem.
     • Example: Tail recursion
       – Take the first (or last) of the n items of information.
       – Make a recursive call to the rest of n-1 items, believing the recursive call will give you the correct result.
       – Given this result and the information you have on the first (or last) item, conclude about the n items.
  3. Just solve this subproblem!
Decomposition of the problem

- You’re all used to the following technique.

```c
int factorial(int n)
{
    int temp = 1;
    for (int i = 1; i <= n; i++)
        temp *= i;
    return temp;
}
```

- \( n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n = \text{factorial}(n-1)! \)
Base Case

• **BELIEVE** factorial(n - 1) will do the right thing.

```c
int factorial(int n)
{
    if (n <= 1)
        return 1;
    int temp = factorial(n - 1) * n;
    return temp;
}
```

• factorial(n) will believe that factorial(n-1) will return the right value.
• factorial(n-1) will believe that factorial(n-2) will return the right value.
• ...
• factorial(2) will believe that factorial(1) will return the right value.
• **AND MAKE** factorial(1) return the right value!
Problem: Permutation

• Print out the permutations of a given vector.

• E.g.
  • [1,2,3] have the following permutations:
    • [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1].

• void permutation(vector<int>& nums, int start);
void permutation(vector<int>& nums, int start) {
    if (start == nums.size() - 1) {
        for(int i=0; i<nums.size(); ++i)
            cout << nums[i] << ' ,';
        cout << endl;
        return;
    }
    permutation(nums, start + 1);
    for (int i=start+1; i<nums.size(); ++i) {
        swap(nums[start], nums[i]);
        permutation(nums, start + 1);
        swap(nums[start], nums[i]);
    }
}
Practice helps

- Recursion is somewhat counter-intuitive when confronted for the first time.
- Just do a lot of practice and you will see some patterns.
- Try finding more examples by googling.

- Again, the key to recursion is to “believe”! Do not try to track the call stack down and see what happens until you really have to.
Template
Template Classes

class Pair {
public:
    Pair();
    Pair(int firstValue, int secondValue);
    void setFirst(int newValue);
    void setSecond(int newValue);
    int getFirst() const;
    int getSecond() const;
private:
    int m_first;
    int m_second;
};

• This class works only with integers.

• Can we make a “generic” Pair class? (Note that typedef does not do the job for us.)
Template Classes

```cpp
template<typename T>
class Pair {
    public:
        Pair();
        Pair(T firstValue, T secondValue);
        void setFirst(T newValue);
        void setSecond(T newValue);
        T getFirst() const;
        T getSecond() const;
    private:
        T m_first;
        T m_second;
};
```

- Here we go.

Pair<int> p1;
Pair<char> p2;
Template Classes

\[
\text{template<typename T, U> class Pair \{} \\
\hspace{1em} \text{public:} \\
\hspace{2em} \text{Pair();} \\
\hspace{2em} \text{Pair(T firstValue,} \\
\hspace{3.5em} \text{U secondValue);} \\
\hspace{2em} \text{void setFirst(T newValue);} \\
\hspace{2em} \text{void setSecond(U newValue);} \\
\hspace{2em} \text{T getFirst() const;} \\
\hspace{2em} \text{U getSecond() const;} \\
\hspace{1em} \text{private:} \\
\hspace{2em} \text{T m_first;} \\
\hspace{2em} \text{U m_second;} \\
\hspace{1em} \};
\]

• More than one type:

Pair<int, int> p1;
Pair<string, int> p2;
Template Classes

```cpp
template<typename T>
void Pair<T>::setFirst(T newValue) {
    m_first = newValue;
}
```

- Member functions should be edited as well.
Template Specialization

• What if sometimes, we want a template class with certain data type to have its exclusive behaviors?
• E.g., define member function uppercase()
  • pair<int> p1;
  • pair<char> p2;
  • We want to allow p2.uppercase();
  • We don’t want to allow p1.uppercase();
Template Specialization

```cpp
template<>
class Pair<char> {
    public:
        Pair();
        Pair(char firstValue,
             char secondValue);
        void setFirst(char newValue);
        void setSecond(char newValue);
        char getFirst() const;
        char getSecond() const;
        void uppercase();
    private:
        char m_first;
        char m_second;
};
```

• Make an exception.

Pair<char> p1;
Pair<int> p2;

p1.uppercase(); (O)
P2.uppercase(); (X)
Template Functions

```cpp
template<typename T>
void swap(T& x, T& y)
{
    T temp = x;
    x = y;
    y = temp;
}
```

- Pretty much the same trick.
- Call the function without `<>`. The types are automatically detected.

```cpp
int x = 2, y = 3;
swap(x, y);

char j = 'c', k = 'm';
swap(j, k);
```
T minimum(const T& a, const T& b) {
    if (a < b)
        return a;
    else
        return b;
}

• When you are not changing the values of the parameters, make them const references to avoid potential computational cost.

Pass-by-value (especially the copy phase) for ADT is slow
STL Containers
STL

- **Standard Template Library**
  - Library of commonly used data structures.
    - `vector` (array)
    - `set` (binary search tree – will learn it soon)
    - `list` (doubly linked list)
    - `map`
    - `stack`
    - `queue`
STL

• A few common functions:
  – .size() .empty()

• For a container that is neither stack nor queue:
  – .insert() .erase() .swap() .clear()

• For list/vector:
  – .push_back() .pop_back()

• For set/map:
  – .find() .count()

• ... and you’ve seen stacks and queues.
#include <list>
using namespace std;

int main()
{
    list<int> a;
    for (int i = 0; i < 10; i++)
        a.push_back(i);
    cout << a.size() << endl;  // prints 10
#include <vector>
using namespace std;

int main()
{

    vector<int> a;
    for (int i = 0; i < 10; i++)
        a.push_back(i);
    cout << a.size() << endl; // prints 10
STL Iterators

• Suppose I want to iterate through elements in a container:

• For an array, you would do:
  
  ```
  int arr[100];
  ...
  for (int i = 0; i < 100; i++)
  {
      cout << arr[i] << endl;
  }
  ```

• But how do we do this for a list or a set?
STL Iterators

- "abstract" way of traversing through elements
- structure<data type>::iterator -- pointer to an element in a container
- .begin() gives you the "first" element in the container
- .end() indicates that the iteration is complete

```cpp
list<int> l;
for (list<int>::iterator it = l.begin(); it != l.end(); it++)
{
    cout << *it << "\"; // Note that ‘*’!!
}
```
begin(), end(), and back()

• **begin()**: return an iterator that points to the first element.

• **end()**: return an iterator that points to the *past-the-last* element
  - *past-the-last*: a theoretical element to represent the place after the last element.

• **back()**: return an iterator that points to the *last* element.
STL Iterators

- If you need to iterate in the reverse direction, you can optionally use `rbegin()` and `rend()`:

```cpp
void func(const list<int> &l)
{
    for (list<int>::const_iterator it = l.rbegin(); it != l.rend(); it++)
    {
        cout << *it; // Note that ‘*’!!
    }
}
```

- Note that you’re still using `it++` to “advance” the iterator.
STL Iterators

• Use **const_iterator** when the container is constant!

```cpp
void func(const list<int> &l)
{
    for (list<int>::const_iterator it = l.begin(); it != l.end(); it++)
    {
        cout << *it << " ";
    }
}
```
STL Iterators

• Iterators are used to call some important functions like insert() and erase():

```cpp
list<int> myList;
myList.push_back(0);  // 0
myList.push_back(1);  // 0 1

list<int>::iterator it = myList.begin();
it++;
myList.insert(it, 30);  // 0 30 1, it still points to 1.
myList.erase(it);       // 0 30
```
Quick Note on `erase()`

- Suppose you’re given a structure and would like to remove all elements that satisfy a certain condition:

```cpp
for (list<int>::iterator it = l.begin(); it != l.end(); it++)
{
    if (*it == 10)
    {
        l.erase(it);  // remove the element pointed by it
    }
}
```

- What is the problem here?
Quick Note on `erase()`

- Suppose you’re given a structure and would like to remove all elements that satisfy a certain condition:

```cpp
define (list<int>::iterator it = l.begin(); it != l.end();)
{
    if (*it == 10)
    {
        it = l.erase(it); // remove the element pointed by it
    }
    else
    {
        it++;
    }
}
```

- `erase()` returns an iterator for the next element.
Insight: List

• How list is implemented: *doubly linked list*.

• No [] allowed to access elements in List.

• Using iterator to traverse a list is always *Safe*.

• And: >, >=, <, and <= comparisons are NOT VALID for list iterators!
Insight: Vector

• How vector is implemented: *dynamic array*.

• We can use [] to access elements in a vector.

• >, >=, <, and <= comparisons are VALID for vector iterators.

• But there might be dangerous behaviors on vector iterators each time we have performed insertion/deletion (incl. push_back()).
int main () {
    vector<int> v;
    v.push_back(50);
    v.push_back(22);
    v.push_back(10);

    vector<int>::iterator b = v.begin();
    vector<int>::iterator e = v.end();
    for (int i = 0; i < 100; i++) {
        v.push_back(i);
    }
    while (b != e) {
        cout << *b++ << endl;
    }
}
Dangerous Behavior of Vector Iterator

• Insertions and deletions on *vectors*, will possibly INVALIDATE any iterators defined on that vector !!!
Dangerous Behavior of Vector Iterator

• Dynamic arrays resize themselves as needed.

• Whenever this happens, the old array is deleted in favor of a new one, but the old iterators are not updated, so they refer to the deallocated memory.

• Insertion at certain point causes the array of vector to expand (new array is created).

• Deletion at certain point causes it to shrink (also creates a new array).
Dangerous Behavior of Vector Iterator

• Reinitialize iterators of a vector whenever its size has been changed.
• *(We don’t need to do that for List)*
## Differences between Vectors and Lists

<table>
<thead>
<tr>
<th></th>
<th>Vector</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>Allowed</td>
<td>Not allowed</td>
</tr>
<tr>
<td>Compare iterators (&lt;, &gt;, =, etc)</td>
<td>Allowed</td>
<td>Not allowed</td>
</tr>
<tr>
<td>Use iterators after modifying contents</td>
<td>Not safe. Iterators need to be reinitialize</td>
<td>Safe</td>
</tr>
<tr>
<td>Body container</td>
<td>Dynamic Array</td>
<td>Doubly Linked List</td>
</tr>
</tbody>
</table>
STL

• You don’t have to memorize names of member functions for each – you can just look things up when you need to.
  e.g. http://www.cplusplus.com/reference/stl/

• But **do** remember:
  – what data structure each container implements
  – how to use iterators
Bugs in your software are actually special features :)}