Sira Songpolrojjanakul

Standard Template Library (STL) Container II

ตัวเก็บเชิงเส้น (Sequence containers)

- vector (dynamic array)
- list (doubly linked list)
- forward_list (singly linked list) [C++11]
- deque (2 way queue)
- array (static array) [C++11]

ความสำคัญโดยสังเขป

- **vector สำคัญมาก ใช้เวลาเก็บกราฟ**
- list ได้ใช้น้อยมาก แต่ก็ต้องรู้
- forward_list เป็นลิสต์ทางเดียว แทบไม่เห็นว่าใช้ ใช้ในกรณีใช้ list แล้ว time out
- deque ระบุว่าใช้กับ sliding window แต่ในความจริงใช้ queue ก็ทำได้ นั่นคือในทางปฏิบัติแทบไม่เห็นว่าใช้
- array เป็น static array ของ STL เหมือน array C ปกติ ซึ่งใช้อาเรย์ C ดีกว่า

- ในที่นี้ จะกล่าวถึงเฉพาะ vector และ list เฉพาะที่สำคัญเท่านั้น
- ดูข้อมูลของ container ตัวอื่น และฟังก์ชันเพิ่มเติมได้ที่ http://www.cplusplus.com/reference/stl/

เวกเตอร์ (vector)

- เหมือนอาเรย์ C
- แต่มีความสามารถในการขยายขนาดได้

การประกาศ

- #include<vector>
- ประกาศเวกเตอร์ vector<**type**> **ชื่อ**;
- ฟังก์ชันที่สำคัญพื้นฐาน
- v.push_back(x) // ใส่ x ต่อด้านท้าย
- v.pop_back() // เอาตัวท้ายสุดออก
- v.clear() // ลบค่าทุกตัวใน v ออก
- v.size() // return ขนาดของ v
- v.empty() // return true ถ้า v ว่าง

การใช้งาน

- รับค่าลงอาเรย์
- รับ N ตามด้วยจำนวนเต็ม N ตัว (N<=100000)
- ปกติที่ทำ

```
int arr[100005],N;
scanf(“%d”,&N);
for(int i = 0;i<N;i++)
    scanf(“%d”,&arr[i]);
```

การใช้งาน

- ใช้ vector

```
int N,k;
vector<int> arr;
scanf(“%d”,&N);
for(int i = 0;i<N;i++)
{
    scanf(“%d”,&k);
    arr.push_back(k);
}
```

ดีกว่าอย่างไร

- ข้อดี
- ไม่ต้องระบุจำนวนช่องเอง นั่นคือไม่มีปัญหาเกี่ยวกับการจองอาเรย์ไม่พอ
- ข้อเสีย
- ช้ากว่า (แต่โดยทั่วไปก็ไม่มากอะไรนัก)

เกิดอะไรขึ้น

```
vector<vector<int>> v;
```

Compile error หรือ รันได้ปกติ

สังเกตที่ >>

ในกรณี C++98 มันจะตีความหมายผิดเป็นเครื่องหมายดำเนินการ ต้องพิมพ์เว้นวรรคระหว่าง >> เป็น

```
vector<vector<int> > v;
```

แต่ถ้าใช้ C++11 จะคอมไพล์ได้

ประกาศ vector แบบกำหนดจำนวนช่อง

```
vector<int> v(10);
```

มีความหมายเหมือนกับ `int v[10];`

และใช้งานได้เหมือนกันเลข

สังเกตว่า ใช้ () ในการระบุจำนวนช่องของ vector

หากใช้ `vector<int> v[10];`

จะได้เป็นเวกเตอร์ 2 มิติแทน

ประกาศ vector แบบกำหนดจำนวนช่อง

```
vector<int> v(10);
```

หากใช้ `v.push_back(111);`

สิ่งที่เกิดขึ้นคือ `v[10]=111;`

กล่าวคือ มันเอา `111` ไปต่อท้ายจากที่จองจำนวนช่องไว้

หากใช้ `v.size()` จะได้ `11`

แก้ขนาด vector

```
v.resize(ขนาด);
```

เช่น

```
vector<int> v(5);
v[0]=7;
v.resize(10);
```

จะได้ว่า `v` มี `10` ช่อง `[0..9]` โดยค่าที่ใส่ไปแล้วไม่หายไป (เว้นแต่จะ `resize` ให้เล็กลงส่วนที่ถูกตัดออก จะหายไป)

กำหนดค่าเริ่มต้น

```
vector<int> v(20,1);
```

จะได้ `v[0..19]` ที่ทุกช่องมีค่า `1`

```
vector<string> sch(20, “SK”);
```

จะได้ `sch[0..19]` ที่ทุกช่องมีค่า `“SK”`

คัดลอกค่าใน vector

กรณีอาเรย์ C ต้อง loop assign ค่า

```
int A[127],B[127];
for(int i=0;i<127;i++)
    B[i]=A[i];
```

กรณี vector สามารถใช้ = ได้เลย

```
vector<int> A,B;
…
B=A;
```

หลากหลายวิธีคัดลอกค่าใน vector

```
vector<int> v1;
// ...
vector<int> v2 = v1;
vector<int> v3(v1);
vector<int> v4(v1.begin(), v1.end());

int arr[]={0,1,2,3,4};
vector<int> vv(&arr[0],&arr[5]);
vector<int> vv(arr, arr+5);
```

ระบุตำแหน่งใน vector/array

บางครั้ง จำเป็นต้องระบุ**ตำแหน่ง**ในอาเรย์หรือเวกเตอร์ เพื่อทำอะไรบางอย่าง อย่างในหน้าที่แล้ว ก็มีวิธีการระบุตำแหน่ง เพื่อทำการบอก `constructor` ว่า ต้องการคัดลอกค่าจากไหนถึงไหน

ระบุตำแหน่งใน vector/array

การระบุตำแหน่งมี 4 แบบ

1. `v.begin(), v.end()`
   - `v.begin()` ชี้ที่ตัวแรกของ `v`
   - `v.end()` ชี้ถัดจากตำแหน่งสุดท้ายของ `v` ไปหนึ่งตัว
   - `array C` ใช้วิธีนี้ไม่ได้
2. `v.begin(), v.begin()+1, …, v.begin()+N`
3. `&v[0], &v[1], …, &v[N]`
4. `v, v+1, …, v+N`
   - ชี้ตัวที่ `0,1,…,N`

2 แบบบนใช้กับ `vector` เท่านั้น

ระบุตำแหน่งใน vector/array

สังเกตว่า ตอนคัดลอกค่า ตัวท้ายต้องประกาศเกินมา `1` ตัว

เพราะว่า ฟังก์ชันของ `STL` ถ้าระบุเป็นช่วง มันจะทำบนช่วง `[x,y)`

นั่นคือ ตัวที่ `y` ไม่ถูกทำ แต่ตัวที่ `y-1` ถูกทำ

vector 2 มิติ

```
vector< vector<int> > Matrix;
```

จะได้ `Matrix[][]`

สังเกตว่าถ้าต้อง `push_back` จะต้อง `push_back` เป็น `vector` เข้ามา

ดังนั้นกำหนดขนาดเลยดีกว่า เพื่อให้เรียกช่อง `[x][y]` ได้

```
vector< vector<int> > Matrix(N,
vector<int>(M,-1));
```

ได้เมทริกซ์ขนาด `N*M` โดยทุกช่องมีค่าเป็น `-1`

vector 2 มิติ

```
vector< int > Matrix[N];
```

จะได้ `Matrix[][]`

แต่ไม่สามารถกำหนดค่าเริ่มต้นหรือขนาดเลขได้ แบบหน้าที่แล้ว

```
vector< int > Matrix[N](M,-1);
```

ต้อง `loop` กำหนดขนาดและค่าเอง

```
for(int i=0;i<N;i++)
    Matrix[i].assign(M,-1);
```

vector 2 มิติ

ปกติ ถ้าหากต้องการใช้ อาเรย์ 2 มิติ แนะนำให้ใช้อาเรย์ C ปกติดีกว่า เนื่องจากสามารถจัดการได้ง่ายกว่ามาก

ยิ่งถ้าใช้เป็น 3 มิติ หรือมากกว่า การประกาศเป็น vector ยิ่งลำบาก

สรุปคือ หากต้องการใช้อาเรย์หลายมิติ ใช้อาเรย์C ดีกว่า

สรุปเล็กน้อย ก่อนลุยต่อ

clear(), assign(), constructer, copy ล้วนแต่ใช้ O(N)

การระบุตำแหน่ง ทั้ง v[x], &v[x], v.begin(), v.end() ใช้ O(1)

push_back(x) และ pop_back() ใช้เวลาถัวเฉลี่ย O(1)

ว่าง่ายๆ vector ก็ใช้เวลาไม่ได้ช้ากว่าอาเรย์ C ปกติมากนัก

เขียนฟังก์ชันเอง ส่วนของ return type

```
void funt() {…}
```
ฟังก์ชันนี้ไม่ return ค่า

```
int abs(int x){ return x>=0? x:-x; }
```
ฟังก์ชันนี้ return int

แน่นอน ให้ฟังก์ชัน return เป็น vector ก็ได้

```
vector<int> f() { … return v; }
```

เขียนฟังก์ชันเอง ส่วนของ argument

ค่าที่ส่งเข้าฟังก์ชันมีได้ สองแบบ คือ `pass by value` กับ `pass by referance`

สองแบบนี้ต่างกันอย่างไร

Pass by value

คัดลอกค่าจากพารามิเตอร์เข้าฟังก์ชัน
นั่นคือ หากมีการแก้ค่าในฟังก์ชัน
ค่าของตัวแปรเดิมจะไม่มีผลอะไร

ตัวอย่าง

```
void f(int x) { x++; }
int main()
{ int s=5; f(s); } //s ใน main ก็ยังเป็น 5
```

Pass by value

ทำแบบนี้ได้ไหม

```
void f(vector<int> x) { … }
```

คำตอบ

ได้

แต่อย่างที่บอกว่า มันคือการ คัดลอกค่า มาให้กับฟังก์ชันนี้

นั่นคือ จะเสียเวลาเป็น `O(N)` ในการคัดลอกค่า

ดังนั้น ต้องพิจารณาให้ดี ว่าจะทำให้โปรแกรมมีประสิทธิภาพแย่จนรับไม่ได้หรือไม่

Pass by reference

บอกให้ตัวแปรชี้ไปที่พารามิเตอร์ที่เรียก

นั่นคือ หากมีการแก้ค่าในฟังก์ชัน

ค่าของตัวแปรเดิมจะเปลี่ยนด้วย

ให้ใส่ `&` ที่หน้า `argument` ของฟังก์ชัน

Pass by reference

```
void swap(int &x, int &y)
{
   int temp;
   temp = x; /* save the value at address x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */

   return;
}

int main()
{
…
 swap(a, b);
…
}
```

Pass by reference

หากต้องการส่ง `vector` เข้าฟังก์ชัน โดยวิธีนี้ก็ใช้

```
void funt(vector<int>& v)
```

หากต้องการส่ง `vector` เข้าไปในฟังก์ชัน โดยเพิ่มเงื่อนไขว่า แก้ไขค่าใน `vector` ที่ส่งมานี้ไม่ได้ ให้ระบุ `const` เพิ่ม

```
void funt(const vector<int>& v)
```

Pass array to function

ส่ง อาเรย์เข้า ฟังก์ชัน ใช้หัวฟังก์ชันได้ 2 แบบ คือ

```
void funt(int *param)
void funt(int param[])
```

สองแบบบนนี้ ไม่ได้ระบุขนาดอาเรย์ที่ส่งเข้าฟังก์ชัน

```
void funt(int param[10])
```

ระบุขนาด

Sample code

```
double getAverage(int arr[], int size)
{
  int    i, sum = 0;  double avg;

  for (i = 0; i < size; ++i) sum += arr[i];
  avg = double(sum) / size;

  return avg;
}
int main ()
{
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;

   // pass pointer to the array as an argument.
   avg = getAverage( balance, 5 ) ;

   // output the returned value
   cout << "Average value is: " << avg << endl;

   return 0;
}
```

Remark

ปกติ เวลาเขียนโปรแกรมเชิงแข่งขัน จะไม่นิยม `pass by referance` หรือส่งอาเรย์เข้าฟังก์ชัน นิยมเขียนตัวแปรนั้นหรือ `array` นั้นเป็น `global` ไปเลยมากกว่า

ตัวอย่าง pair อีกนิด

```
pair<string, pair<int,int> > P;
string s = P.first;
int x = P.second.first;
int y = P.second.second;
```

ฟังก์ชันอื่นๆ พื้นฐาน

include

```
#include<algorithm>
```

sort(st,ed)

เรียงข้อมูลจากน้อยไปมาก

```
sort(v.begin(),v.end());
sort(v,v+N); // N=size of array
sort(&v[0],&v[N]);
```

สิ่งที่ได้

อาเรย์/vector v จะเรียงจากน้อยไปมาก

O(n log n)

หวังว่ายังจำได้ว่าสามารถใช้การบอกตำแหน่งได้ 3 แบบ

reverse(st,ed)

กลับอาเรย์

```
reverse(v.begin(),v.end());
```

สิ่งที่ได้ มันจะกลับอาเรย์ให้

เช่น

```
string s="12345";
reverse(s.begin(),s.end());
printf("%s",s.c_str); // 54321
```

O(n)

ช่วงคั่นเวลา -> string

string ในที่นี้เป็น datatype ใน c++ แบบหนึ่ง
string c คือ char[]
string c++ ช้ากว่า string c
แต่ถ้าจะใช้ string กับ STL ต้องใช้ string c++
ต่อสตริง ใช้เครื่องหมาย + ได้
เช่น string s1="ab",s2="cd";
s1+=s2; // s1="abcd"

ช่วงคั่นเวลา -> string

สามารถเอา `string c` มาใส่ได้เลย

```
char str[]="123";
string s=str;
```

เวลารับเข้า `string c++` ใช้ `%s` ได้

แต่เวลาพิมพ์ ถ้าใช้ `%s` ต้องเพิ่ม `.c_str` ท้ายตัวแปรด้วย จึงจะใช้ได้

แบบตัวอย่างที่เขียนให้ดูในหน้าที่แล้วๆ

ช่วงคั่นเวลา -> string

เปรียบเทียบ สตริงได้ตรงๆ เลย

```
string s1="abc",s2="aaa";
X = s1 < s2; // X=false
```

string c++ ก็ยังมีลักษณะเป็น array คือเข้าถึงด้วย index ได้

```
s1[2]='c';
```

min(x,y) / max(x,y)

return ค่ามากหรือค่าน้อย ตามชื่อของฟังก์ชัน

```
x=min(1,2);     // x=1
y=max(1,2);     // y=2
z=min(max(2,3),1);     // z=1
```

O(1)

next_permutation(st,ed)

หา permutation ตัวถัดไป

return true ถ้ามี permutation ตัวถัดไป
return false ถ้าไม่มี permutation ตัวถัดไป
แล้ว

หาpermutation ทุกตัว O(n!)
หาตัวเดียว O(n)

```
// 
#in
#in

int
  i

  s

  s
  d

myi
  }

  s
' '

  return 0;
}
```

```
The 3! possible permutations with 3 elements:
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
After loop: 1 2 3
```

prev_permutation(st,ed)

หา `permutation` ตัวก่อนหน้า
คล้ายกับ `next_permutation`

```
// next_permutation example
#include <iostream>     // std::cout
#include <algorithm>    // std::next_permutation, std::sort, std::reverse

int main () {
  int myints[] = {1,2,3};

  std::sort (myints,myints+3);
  std::reverse (myints,myints+3);

  std::cout << "The 3! possible permuta
  do {
    std::cout << myints[0] << ' ' << my
  } while ( std::prev_permutation(myint

  std::cout << "After loop: " << myints
myints[2] << '\n';

  return 0;
}
```

```
3 2 1
3 1 2
2 3 1
2 1 3
1 3 2
1 2 3
After loop: 3 2 1
```

lower_bound(st,ed,val)

return ตำแหน่ง (iterator/pointer) ของ **ค่าที่น้อยที่สุดที่มากกว่าหรือเท่ากับตัวที่หา**

หากไม่มี return v.end()

Array/vector ที่จะทำต้องเรียงจากน้อยไปมากแล้ว

O(log n)

upper_bound(st,ed,val)

return ตำแหน่ง (iterator/pointer) ของ **ค่าที่น้อยที่สุดที่มากกว่าตัวที่หา**

หากไม่มี return v.end()

Array/vector ที่จะทำต้องเรียงจากน้อยไปมากแล้ว

O(log n)

```
// lower_bound/upper_bound example
#include <iostream>     // std::cout
#include <algorithm>    // std::lower_bound, std::upper_bound, std::sort
#include <vector>       // std::vector

int main () {
  int myints[] = {10,20,30,30,20,10,10,20};
  std::vector<int> v(myints,myints+8);           // 10 20 30 30 20 10 10 20

  std::sort (v.begin(), v.end());                // 10 10 10 20 20 20 30 30

  std::vector<int>::iterator low,up;
  low=std::lower_bound (v.begin(), v.end(), 20); //          ^
  up= std::upper_bound (v.begin(), v.end(), 20); //                   ^

  std::cout << "lower_bound at position " << (low- v.begin()) << '\n';
  std::cout << "upper_bound at position " << (up - v.begin()) << '\n';

  return 0;
}
```

```
lower_bound at position 3
upper_bound at position 6
```

Need to know

`vector<int>::iterator up;`

สิ่งนี้เรียก `iterator` เป็นตัวชี้ใน `stl`

จากหน้าที่แล้ว

`up = upper_bound (v.begin(), v.end(), 20);`

`up` จะได้ตำแหน่ง `[0x....]`

`up-v.begin()` จะได้ `index` ที่ `up` ชี้ `[6]`

`*up` จะได้ค่าที่ตำแหน่งนั้น `[30]`

Need to know

ใช้ pointer ก็ได้

```
int *up;
```

```
up = upper_bound (&v[0], &v[n], 20);
```

up จะได้ตำแหน่ง [0x....]

up-v จะได้ index ที่ up ชี้ [6]

*up จะได้ค่าที่ตำแหน่งนั้น [30]

สังเกตว่า ต่างกันที่การระบุ index ใน v และใช้ pointer กรณีเราจะหาในอาเรย์

Loop ใน vector

Loop เลขทุกตัวในเลข vector ทำอย่างไร

```
vector<int> v;
for(int i=0;i<v.size();i++)
    printf("%d\n",v[i]);
```

Loop ใน vector

ยังมีวิธีลูปแบบอื่น ใช้ iterator

```
vector<int> v;
vector<int>::iterator it;
for(it=v.begin();it!=v.end();it++)
    printf("%d\n",(*it));
```

Loop ใน vector

พิจารณา

```
vector<pair<int,pair<int,int> > >  x;
```

Loop ด้วย i ก็ง่ายๆ

แต่ถ้าจะลูปด้วย iterator ก็ต้องประกาศ iterator ชนิดเดียวกันกับสิ่งที่จะไปลูป

จะเห็นว่า loop ด้วย i ดีกว่า

จริงเหรอ ?

iterator

คำตอบ

จริง

อย่างไรก็ดี ก็ไม่สามารถหลีกเลี่ยงไม่ใช้ `iterator` เลยไม่ได้
อย่างจะใช้ `lower_bound` ก็ต้องใช้ `iterator`

More information

ยังมีฟังก์ชันอื่นๆ ของ `algorithm`
ดูได้ที่
http://www.cplusplus.com/reference/algorithm/

แต่ที่ใช้บ่อยๆ ก็มีเท่านี้แหละ

Motivation

ต้องการแทรกข้อมูลลงไปตรงกลางอาเรย์ ทำได้ไหม อย่างไร

```
0 1 2 3 4 5
```

ต้องการแทรก `x` หลังตำแหน่ง `3 (0 base index)`

```
0 1 2 3 x 4 5
```

ทำได้ แต่เสียเวลา O(n)
ก็คือ ไปที่ตำแหน่ง 3
แล้วก็ขยับตัวหลังตำแหน่งที่ 3 ไปหนึ่งตำแหน่ง
แล้วใส่ x ลงไป

```
0 1 2 3 4 5 - - -
0 1 2 3 - 4 5 - -
0 1 2 3 x 4 5 - -
```

ใช้ `insert` ของ `vector` ดีไหมนะ

```
v.insert(it,val);
```

ใส่ที่ตำแหน่ง `it`

```
// inserting into a vector
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> v;
  std::vector<int>::iterator it;
  for(int i=0;i<6;i++)
     v.push_back(i);

  it=v.begin();
  v.insert(v.begin()+4,9);

  for (it=v.begin(); it<v.end(); it++)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

```
0 1 2 3 9 4 5
```

ก็ยัง `O(n)` อยู่ดี

ถ้าอยากให้ `insert O(1)` ต้องใช้ `list`

To be continued…

ตัวอย่างตอนต่อไป

- พบกับภาชนะที่ `insert/delete` ตรงไหนก็ `O(1)`
- `Array` ที่ `index` เป็นอะไรก็ได้
  - `mp[2000000000]=1;`
  - `m[“suankularb”]=“SK”;`
- ภาชนะที่ข้อมูลในนั้นเรียงอยู่เสมอ `->` เอาไปทำอะไรนะ

จบเนื้อหา

สวัสดี