Lecture 5 : 배열과 벡터 (Array and Vector)

배열은 가장 많이 쓰이는 보편적이며 대표적인 자료구조이다. 그러나 C언어에서 제공하는 배열(C-style array)의 가장 큰 단점은 compile time 이전에 그 크기가 결정되어야 한다는 것이다. 즉 사용자의 입력을 받아서 실행 도중에 크기를 동적으로(dynamic) 결정할 수 없어 유연성이 크게 떨어진다. 예를 들어 사용자로부터 N을 받아서 크기 N인 배열을 run time에 확보하는 하는 것을 불가능하다.[1)]

만일 사용자에게 편리한 동적 변화가 가능한 array 구조를 활용하려면 system call을 사용하는 malloc( ) 등을 이용하여 복잡하게 메모리를 할당받아서 그 indexing을 직접 사용자가 관리해야 한다. 이런 제한을 설정한 이유는 compile time에 배열의 크기를 결정하면 특정 원소의 위치를 미리 결정할 수 있어 수행 속도를 높일 수 있기 때문이다. 그러나 실제 현장에서 dynamic allocation을 할 수 없는 제한은 꽤 심각한 결점이 아닐 수 없다. STL에서는 이 문제를 해결하기 위하여 array의 속도와 dynamic allocation을 가능한 변형 구조인 벡터(vector)를 제공하고 있다. 요약하면 다음과 같다.

C-style array : 크기가 정해진 물통. 오래된 재료를 사용하여 만든
STL array : 사용자 편의 기능이 부착된 물통
STL vector : 한 쪽 방향으로 크기를 늘일 수 있는 “자바라형” 물통

STL deque : 양쪽 방향으로 늘일 수 있는 물통

5.1 배열 **Array : Random Access machine**의 기능을 극대화한 구조.

vector와 다르게 크기를 알 수 있는 경우 활용하면 성능이 더 뛰어나다. 퍼포먼스가 중요한 실제 애플리케이션에서 벡터의 문제점은 다음과 같다.

a. 생성과 소멸을 하는데 필요한 “비용(cost)”이 만만치 않다. 이 오버헤드는 특히 사용해야 하는 벡터의 수가 많으면 많을수록 증가한다.

b. 전체 구조의 조정(크기 변화)이 런타임(run time)중에 실행하므로, 즉 벡터는 동적 배열이기 때문에 크기를 변화시키는 동작, 예를 들어 원소 삽입(insert)나 특정 원소 삭

1) 이것을 “야매”로 하려면 memory allocation malloc( ) 을 사용하는 것이다.

제(remove)를 빈번하게 한다면 전체적인 성능은 떨어진다. 물론 코딩을 하는 프로그래머 입장에서 이 작업이 몇 줄의 간단한 함수 호출로 가능하지만 이것을 처리해주는 기계 입장에서는 큰 부담이 된다. 코딩이 간단하다고해서 그것이 CPU에서 동작도는 수행(execution)까지 간단해지는 것은 아니기 때문이다.

c. 동적으로 증가하거나 감소하는 경우 불필요한 벡터 전체 단위의 복사가 일어난다. 이 부분은 C++11 에서 move semantics라는 개념에 의해 극복되었다.

배열(array)을 규정하는 두 원칙은 다음과 같다.

i) 동일한 형식의 동일한 물리적 크기의 단위 데이터[2]들의 집합체(collection)
ii) index parameter에 의해서 물리적으로 인접하게 저장.

기존의 배열은 위에서 언급한 벡터의 오버헤드가 없다. 컴파일을 시작할 시간 즉 compile time에 배열의 확정된 크기를 결정하기 때문에 추가의 부담은 있을 수 없기 때문이다. 그러나 벡터가 가져오는 장점은 당연히 포기해야 한다. 배열 이름이 그 첫 번째 원소를 가리키는 포인터로 묵시적인 형(type) 변환이 허용되므로, 보안상의 문제 또한 있다. 이런 문제를 해결하기 위해 C++11에서는 기존의 컨테이너와 유사하면서 정적인 배열을 선언하기 위해 std::array가 채택되었다. 이를 통해 벡터의 장점은 그대로 취하는 한편, 정해진 크기의 정적 배열을 만들 수 있다.

std::array의 좋은 점 가운데 하나는 벡터에서와 같이 상수시간에 random access 가 가능하다. 또한, 메모리 측면에서 볼 때, 벡터가 free-store 라고 불리는 영역에 엘리먼트를 비연속적으로 배치하여 사용하는 것과는 다르게 std::array는 스택 공간에 연속적으로 엘리먼트를 배치한다. 또한 std::array는 다른 container와 마찬가지로 생성자, 소멸자, 복사생성자, 대입 연산자를 지원하는 것이다. 따라서 우리는 C-style의 배열대신 C++11 표준으로 승인된 std::array를 본격적으로 사용해야 한다.

배열과 벡터를 공부할 때 반드시 익혀야 할 내용은 다음과 같다.

f.1. 어떤 것이 코딩에 더 편한가 ?
f.2. 저장과 추출(retrieval)[3]이 편한가 ?
f.3. 큰 크기의 데이터에 대하여 문제없이 동작하는가 ?
small, med. large, huge에 대하여 성능의 차이가 없는가
f.4 실제 차지하는 메모리의 물기적인 크기는 얼마인가 ?
(어떻게 자료구조를 구성하였길래 그러한가 ?)

Q) Python에서 다음의 크기를 측정해보자.

---

2) python의 list는 서로 다른 유형의 객체를 넣을 수 있다. C++에서 이것을 가능하게 해주는 것은 tuple이다.
3) retrieve, 뭔가를 가서 가져오는 동작을 의미한다. 골든 리트리버라는 개가 이것을 잘한다.

```
import sys
md={ }
md["good"]=1234

La = [ True , 'a', 100, "summer", "summer-time",
      3.145,    31.1456789,
      [], [4], [4,5],\
      [4,5,6], [4,5,6,7],
      [[]], 2+3j,
      "대", "대한", "대한뽕", \
      (2,3), md, range(500) ]

for w in La :
   print(  type(w), ": ",  sys.getsizeof(w), "  ", w )
```

getsizeof( )

```
<class 'bool'> :    28    True
<class 'str'> :     50    a
<class 'int'> :     28    100
<class 'str'> :     55    summer
<class 'str'> :     60    summer-time
<class 'float'> :   24    3.145
<class 'float'> :   24    31.1456789
<class 'list'> :    56    []
<class 'list'> :    64    [4]
<class 'list'> :    72    [4, 5]
<class 'list'> :   120    [4, 5, 6]
<class 'list'> :   120    [4, 5, 6, 7]
<class 'list'> :    64    [[]]
<class 'complex'> :  32    (2+3j)
<class 'str'> :      76    대
<class 'str'> :      78    대한
<class 'str'> :      80     대한뽕
<class 'tuple'> :   56     (2, 3)
<class 'dict'> :   232    {'good': 1234}
<class 'range'> :   48    range(0, 500)
```

```
#include <iostream>     //cout
#include <tuple>        //tuple,get,tie,ignore
using namespace std ;

int main () {

 tuple <int,char> foo (10,'x');
 auto bar = make_tuple ("test", 3.1, 14, 'y');

 get<2>(bar) = 100;           // access element

 int myint; char mychar;

 tie (myint, mychar) = foo;         // unpack elements
// unpack (with ignore)
 tie (ignore,ignore, myint, mychar) = bar;

 mychar =get<3>(bar);

 get<0>(foo) =get<2>(bar);
 get<1>(foo) = mychar;

 cout << "foo contains: ";
 cout << get<0>(foo) << ' ';
 cout << get<1>(foo) << '\n';

  return 0;
}
```

Q) 어두운 방안에 10개의 소주병이 일렬로 배열되어 있다. 이중 7번째 소주병에는 값비싼 양주가 들어있다. 단 여러분에게 소주병을 잡아낼 수 있는 기회는 단 1번. 여러분은 이 일을 할 수 있을 것인가 ?

Q) Python에서 List와 array의 차이를 설명

5.2 대부분 언어는 배열(array)을 기본적으로 제공된다.

a) C-style array `float myf[100] ;`

- 이미 크기가 결정되어야 한다.
- 항상 크기를 염두에 두어야 한다.
- index를 넘어가서 access하면 난리가 난다.
  그 유명한 segmentation fault error ! 를 만나게 될 것이다.

b) C++ array

```
array <int,100> Good ;
Good.begin( ), Good.end( )
```

c) Python array - 파이썬에서도 배열을 제공한다. List의 편의성과 속도, 공간의 문제를 모두 해결해 줄 수 있는 대안이 될 수 있다. 하지만 실제 사용해보면..

```
"""
Commonly used type codes are listed as follows:

Code    C              Python     Min bytes

b   signed char            int         1
B   unsigned char    int         1
u   Py_UNICODE       Unicode     2
h   signed short           int         2
H   unsigned short   int         2
i   signed int       int         2
I   unsigned int           int         2
l   signed long            int         4
L   unsigned long    int         4
f   float            float       4
d   double           float             8
"""

import array as arr
a = arr.array('d', [1.1, 3.5, 4.5])


b = arr.array('B', [11, 12, 23, 45, 0 ] )

c =arr.array('l')
d =arr.array('u', 'hello \u2641')
e =arr.array('l', [1, 2, 3, 4, 5])
f =arr.array('d', [1.0, 2.0, 3.14])
```

5.3 vector는 STL의 가장 기본적인 보관(container) 자료구조이다.

- 벡터(vector) = 한쪽이 열린 array
  - 시스템은 피곤하지만, 프로그래머는 편하다.
- C++에서 보편적으로 사용되는 배열(array)에 대응하는 자료구조이다.
- 벡터의 다양한 built-in 연산을 사용하면 빠르고 정확하게 작성할 수 있다.
- 따라서 이미 구현되어 있는 연산을 잘 숙지하고 있어야 한다.

  예) `max_element( ), min_element( )`

  예) `for(i=1 ;.. )` 이렇게 몸으로 때우면 안된다. 절대로...

5.4 Vector의 기본적인 member function

vector를 잘 사용하고 효율적으로 활용하는 지름길은 다양한 member function을 활용하는 것이다. 단 하나의 member fucntion으로 할 수 있는 작업은 직접 만들거나 또는 다른 member function을 이용해서 작성하는 것은 반드시 피해야 한다.

- `empty( )`
- `size( )`, 단 이것은 `sizeof(v)`와는 다름에 유의해야 한다.
- `capacity( )`
- `max_size( )`
- `reserve( )` ; 매우 중요한 operation 이다.

resizing에 의한 시간 손실을 막아줄 수 있다.

vector::assign	vector::end
vector::at	vector::erase
vector::back	vector::front
vector::begin	vector::get_allocator
vector::capacity	vector::insert
vector::cbegin	vector::max_size
vector::cend	vector::operator=
vector::clear	vector::operator[ ]
vector::crbegin	vector::rbegin
vector::crend	vector::rend
vector::data	vector::reserve
vector::emplace	vector::resize
vector::emplace_back	vector::shrink_to_fit
vector::empty	vector::size
vector::pop_back	vector::swap
vector::push_back	

예를 들어 아래 코드를 참조하자. 아래는 vector에서 특정 원소를 다른 원소로 바꾸는 replace 함수이다. 이 함수를 알고있지 못할 경우 이 작업은 직접 코딩하기란 쉽지 않다.

```
#include <iostream>     // std::cout
#include <algorithm>    // std::replace
#include <vector>       // std::vector

using namespace std ;
#define vout(msg,x)   \
       cout<<"\n"<<msg; for(auto s:x) cout << "  " << s ;

int main () {
  int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };
  vector<int> myvector (myints, myints+8);
  vout(" replace 전 myvector[]=", myvector ) ;

  cout << " 20을 99로 바꿈." ;
  replace (myvector.begin(), myvector.end(), 20, 99);
  vout(" replace 후 myvector[]=", myvector ) ;
  return 0;
}
```

5.5 Practice-01 : Vector에서의 추가(push_back( )) (Lab) - 시간측정

- 10개짜리 vector를 선언해서 추가로 push_back( ) 으로 10만개를 추가해본다.
- pop_back( )을 이용해서 추가된 10만개를 제거한다.

5.6 Practice-02 : Vector에서의 임의 위치에 삽입(insert( )) - 시간측정

- Vector(array)에서의 중간 삽입은 매우 비싼 연산이다. 왜 그럴까 ?
- 15000개짜리 vector를 만들어서 2번째 원소에 추가하는 작업은 10000번 해본다.

5.7 Practice-03 : Vector에서의 삭제(erase) - 시간측정

- Vector(array)에서의 중간 삭제 역시 매우 비싼 연산이다. 왜 그럴까 ?
- 15000개짜리 vector를 만들어서 특정 위치에서 특정 위치까지 지운다.
- 그리고 크기를 다시 측정해보자.

```
vec_sample.erase( where ) ;
where = find( vec_sample.begin(), vec_sample.end(), 12345 ) ;
vec_sample( where, where + 50 ) ;
```

5.8 Vector에서의 generic algorithm

```
merge(a.begin( ), a.end( ), b.begin( ), b.end( ),
        ostream_iterator<int,char>(cout, " "));

// sorting과 merge의 차이점 : input, sorted data, unsorted data
```

```
sort( vec.begin(), vec.end() ) ;  //  10000개부터 10만개까지
random_shuffle( ) ;
count( ) ;
search( ) ;
max_element( ) ; min_element( ) ;
for_each( ) ;
rotate( ) ;                //어떤 작업을 하는지 실제로 수행시켜보시오.
partition( ) ;             // 둘로 쪼갠다.
```

5.9 Vector의 비교 (제공해주는 이 기능을 활용할 수 있어야 한다.)

- 어떤 학생의 성적평점이 전공필수, 전공선택, 교양, TOEIC 으로 구분되어 있다.
  우리는 이 학생의 성적을 4개의 component가 있는 vector로 구분하여 사용하고자 한다.

- 이 학생들의 등위를 결정하는데 전공필수(R), 선택(S), 교양(C), 영어(E) 순으로 한다.

각 성적으로 0부터 4.5만점의 점수로 한다.

- 데이터의 처음에는 이름(영문자string 최대 10자)와 4개의 floating point가 있다.
- 이들의 등위를 정해서 그 순서대로 출력하는 프로그램을 STL로 구성하시오.

```
Tom  3.27   3.04   2.56   4.0
Mary 3.77   3.45   3.01   2.34
VOX  2.78   3.56   4.09   4.0
PC+  2.56   4.5    3.22   2.5
.....
Emil 1.34   4.35   4.32   4.23
```

■ 지혜의 말씀 5-1: 좋은 질문은 10개의 해답보다 유용하다.

5.10 Vector 비교의 다양한 경우

- 차원이 다른 경우 어떤 결과가 나오는지 생각해보자.

No.	A	B	의미
1	1, 2, 3	4, 5, 6	
2	50, 30, 90	20, 100, 100	
3	'a', 100, 'xyz'	30, 40, 50, 60	
4	1, 2, 3, 4	1, 2, 3, 4, 5	
5	30, 'a'	30	
6	60, 90, 'abc'	12.34, 90.0	
7	'a', 'b', 'c', 'd'	'a', 'b', 'd'	
8	23.67 10.23 -9	23.67 , -9	
9	[1,2], 3, [4,5]	1, [2,3], 4, 5	

5.11 STL vector 자료구조의 측정

- 얼마나 큰 크기가 잡히는지 ? (컴파일러마다 다르다.)
- 다차원으로 얼마까지 잡을 수 있는지 ?
- 100만개의 vector를 scan(read)하는데 걸리는 시간은 ?
- 100만개의 vector를 scan(write)하는데 걸리는 시간은 ?
- 100만개의 vector를 data rotate하는데 걸리는 시간은 ?

5.12 **Vector vs Array : 선택의 기준**

a) 배열의 크기가 더 늘어날 경우가 있는가?
(크기가 줄어드는 것은 심각한 문제는 아니다. 왜 그런지 설명해보시오.)

b) 만일 늘어날 때 그 정도는 어느 정도인가 ?

c) 단위 개체끼리의 Assignment나 swap을 자주 해야 할 필요가 있는가 ?

d) 컴파일러의 특성에 좌우되는가? (sensitivity to compiler)

e) 내부구조가 복잡한가? 예를 들어 단위 element가 또 다른 Class의 일부인가?

f) Operator overloading의 필요성이 있는가 ?

g) 내부 원소의 위치를 자주 바꾸어야 하는가? (List나 tree를 활용)

5.13 STL **vector** 사용시 주의사항

- 코딩된 소스의 길이가 "짧다고" 수행 속도가 빠른 것이 아니다.
- vector<T>의 **첫 위치에(front)** 원소를 insert 할 때
- vector<T>의 **가장 뒤에(back)** 원소를 insert 할 때

```
#include <string>
#include <iterator>
#include <iostream>
#include <algorithm>
#include <array>
#define  allout(MSG,X)  cout<<"\n"<<MSG;for(auto w:X){cout<<" "<<w<<",";}
using namespace std ;

int main() {
    array <int,    3> a1   { {61,52,43} };     // double-braces required
    array <int,    3> a2 = { 11, 22, 33 };    // except after =
    array <string, 4> a3 = { {string("a"), "b", "PNU", "똥강아지"}  };

    allout("a1소팅 전", a1) ;
    sort(a1.begin(), a1.end());
    allout("a1 소팅 후", a1) ;
    allout("a2 처리 전", a2) ;
    reverse( a2.begin(), a2.end() );
    allout("a2 처리 후", a2) ;

    // ranged for loop is supported
    allout("a3의 내용", a3) ;
} // end of main( )
```

```
typedef array<int,4> mytype;

int main() {
    mytype A ;
    array  <int,3>    mya   { {1,2,3} } ;  // double-braces required
    vector < mytype > myb = { {9,8,7}, {-2,-3,-4}, {10,20,30}, {-200,89} } ;

    for(auto& s: mya) // 함수형 표현법, C++에서 가능.
        cout << s << ' ';

    cout << "\n---Next Experiment -------\n" ;
    for(auto& s: myb) // 함수형 표현법, C++에서 가능.
        cout << s[0] << ' ' << s[1] << endl ;

} // end of main( )
```

```
int main () {
  array<int,5> first =  {10, 20, 30, 40, 50}, third ;
  array<int,5> second = {11, 22, 33, 44, 55};

  first.swap (second);

  cout << "first:";
  for (int& x : first) cout << ' ' << x;
  cout << '\n';

  third = first ;
  cout << "second:";
  for (int& x : second) cout << ' ' << x;
  cout << '\n';


  cout << "third:";
  for (int& x : third) cout << ' ' << x;
  cout << '\n';
  return 0;
} // end of main( )
```

```
int main () {

    const char* cstr = "Summer_Time_Killer";
    array<char,30> charray;

    memcpy (charray.data(),cstr,12);
    cout << charray.data() << endl ;

  return 0;
}
```

```
class name {
  public:
    string first;
    string last;
    name(string a, string b){ first = a; last = b; }
    name ()  { first="Cho" ; last="Pig" ; }
    nprint() { cout << "==> " << first << " " << last << endl ;}
};

int main (int argc, const char * argv[]) {
  const int N = 3;
  name T { "Thomas", "Harp" } ;
  array <name, N >  myName ;
  T.nprint() ;
  myName[0].nprint() ;
  myName[1].nprint() ;
  return 0;
}
```