Lecture6: 벡터(6.1절), 리스트(6.2절), 시퀀스(6.3절)

김 강 희
khkim@ssu.ac.kr

요약

- 벡터 (vector) = 배열리스트 (ArrayList)
  - 객체들의 배열을 정의할 때 유용한 STL 클래스(#include <vector>)
  - C++ 배열과 다르게, (1) 동적으로 크기 확장 가능, (2) 삭제시 원소별 소멸자를 자동 호출, (3) 복사시 원소별 생성자 자동 호출
  - 유용한 멤버 함수들을 제공함 (#include <algorithm>)
  - 만약, 벡터 클래스를 사용자가 직접 정의한다면 내부적으로 배열 구현 또는 노드 구현(즉, 링크드 리스트)을 사용할 수 있음
- 리스트 (list)
  - 객체들의 리스트를 정의할 때 유용한 STL 클래스 (#include <list>)로서 doubly linked list로 구현됨
  - 삭제시 원소별 소멸자를 자동 호출, 복사시 원소별 생성자 자동 호출
  - 유용한 멤버 함수들을 제공함 (#include <algorithm>)
- 시퀀스 (sequence)
  - 개념적으로 ArrayList와 NodeList의 집합체(union) 성격의 STL 클래스

6.1절 벡터

- 배열을 이용한 벡터 클래스 구현

```
typedef int Elem;
class ArrayVector {
public:
  ArrayVector();
  int size() const;
  bool empty() const;
  Elem& operator[](int i);
  Elem& at(int i) throw(IndexOutOfBounds);
  void erase(int i);
  void insert(int i, const Elem& e);
  void reserve(int N);
  // ... (housekeeping functions omitted)
private:
  int capacity;
  int n;
  Elem* A;
};
```

벡터 객체 사용 예시

연산	출력	V
insert(0,7)	–	(7)
insert(0,4)	–	(4,7)
at(1)	7	(4,7)
insert(2,2)	–	(4,7,2)
at(3)	"error"	(4,7,2)
erase(1)	–	(4,2)
insert(1,5)	–	(4,5,2)
insert(1,3)	–	(4,3,5,2)
insert(4,9)	–	(4,3,5,2,9)
at(2)	5	(4,3,5,2,9)
set(3,8)	–	(4,3,5,8,9)

ArrayVector 클래스

```
ArrayVector::ArrayVector()
  : capacity(0), n(0), A(NULL) { }

int ArrayVector::size() const
  { return n; }

bool ArrayVector::empty() const
  { return size() == 0; }

Elem& ArrayVector::operator[](int i)
  { return A[i]; }

Elem& ArrayVector::at(int i) throw(IndexOutOfBounds) {
  if (i < 0 || i >= n)
    throw IndexOutOfBounds("illegal index in function at()");
  return A[i];
}
```

ArrayVector 클래스

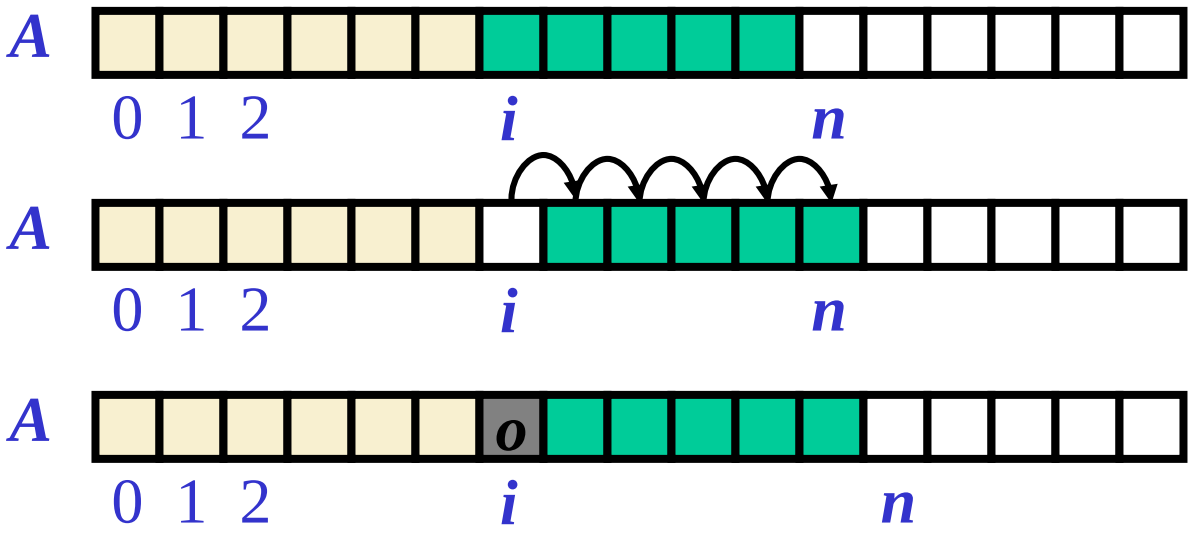
```
void ArrayVector::insert(int i, const Elem& e) {
  if (n >= capacity)
    reserve(max(1, 2 * capacity));
  for (int j = n - 1; j >= i; j--)
    A[j+1] = A[j];
  A[i] = e;
  n++;
}
```

```
void ArrayVector::erase(int i) {
  for (int j = i+1; j < n; j++)
    A[j - 1] = A[j];
  n--;
}
```

```
void ArrayVector::reserve(int N) {
  if (capacity >= N) return;
  Elem* B = new Elem[N];
  for (int j = 0; j < n; j++)
    B[j] = A[j];
  if (A != NULL) delete [] A;
  A = B;
  capacity = N;
}
```

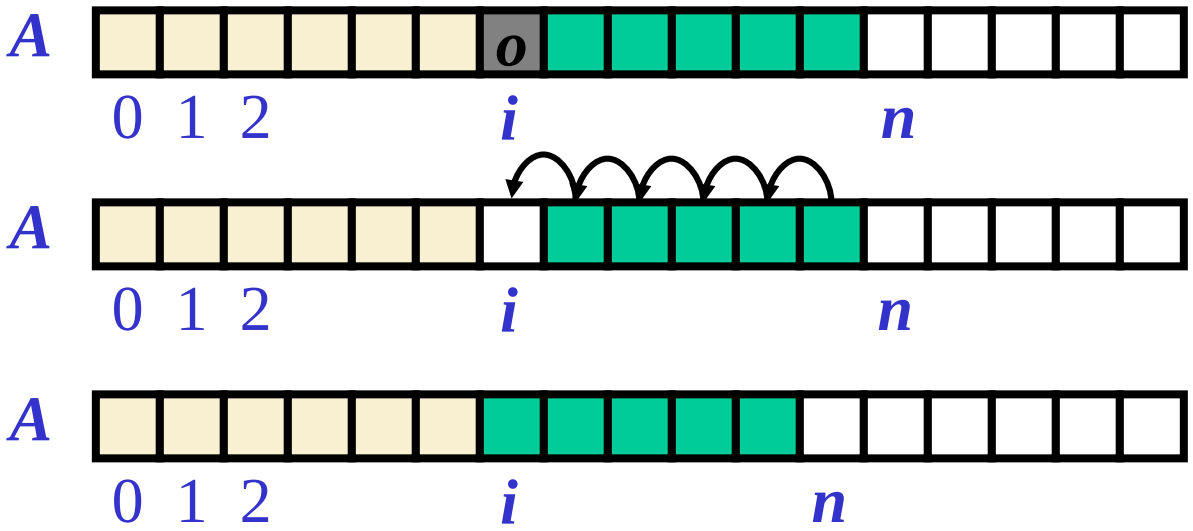
ArrayVector 클래스: insert 동작

- In operation ***insert*(*i*, *o*)**, we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n-1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



ArrayVector 클래스: erase 동작

- In operation ***erase***(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \ldots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



STL Vector 클래스

- vector(n): n개 원소들의 공간을 갖는 벡터 생성
- size(): 원소의 개수
- empty(): 원소가 비어있으면 참, 그렇지 않으면 거짓
- resize(n): 벡터 V의 크기가 n이 되도록 공간 조정
- reserve(n): n개 원소를 저장할 수 있는 공간 선확보
- operator[i]: i번째 원소를 위한 참조 정보 반환
- at(i): V[i]와 같은 의미이나, 인덱스 i가 유효한지 체크
- front(): V의 첫번째 원소의 참조 정보 반환
- back(): V의 마지막 원소의 참조 정보 반환
- push_back(e): 원소 e의 복사본을 V 끝에 추가하고 크기를 1 증가시킴
- pop_back(): 마지막 원소를 제거하고 크기를 1 감소시킴
- 기타 등등

STL Vector 클래스 예제

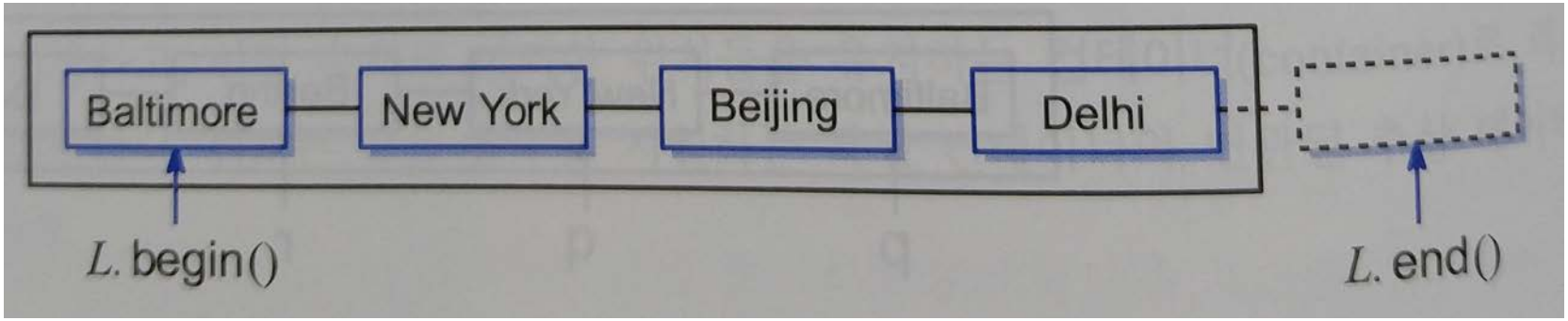
```
int main () {
  int a[] = {17, 12, 33, 15, 62, 45};
  vector<int> v(a, a + 6);                 // v: 17 12 33 15 62 45
  cout << v.size() << endl;                // outputs: 6
  v.pop_back();                            // v: 17 12 33 15 62
  cout << v.size() << endl;                // outputs: 5
  v.push_back(19);                         // v: 17 12 33 15 62 19
  cout << v.front() << " " << v.back() << endl;   // outputs: 17 19
  sort(v.begin(), v.begin() + 4);          // v: (12 15 17 33) 62 19
  v.erase(v.end() - 4, v.end() - 2);       // v: 12 15 62 19
  cout << v.size() << endl;                // outputs: 4

  char b[] = {'b', 'r', 'a', 'v', 'o'};
  vector<char> w(b, b + 5);                // w: b r a v o
  random_shuffle(w.begin(), w.end());      // w: o v r a b
  w.insert(w.begin(), 's');                // w: s o v r a b

  for (vector<char>::iterator p = w.begin(); p != w.end(); ++p)
    cout << *p << " ";                     // outputs: s o v r a b
  cout << endl;
  return EXIT_SUCCESS;
}
```

6.2절 리스트

- 노드를 이용한 리스트 클래스 구현

```
typedef int Elem;                  // list base element type
class NodeList {                   // node-based list
private:
  // Node declaration here...
public:
  // Iterator declaration here...
public:
  NodeList();                      // default constructor
  int size() const;                // list size
  bool empty() const;              // is the list empty?
  Iterator begin() const;          // beginning position
  Iterator end() const;            // (just beyond) last position
  void insertFront(const Elem& e);        // insert at front
  void insertBack(const Elem& e);     // insert at rear
  void insert(const Iterator& p, const Elem& e); // insert e before p
  void eraseFront();               // remove first
  void eraseBack();                // remove last
  void erase(const Iterator& p);     // remove p
  // housekeeping functions omitted...
private:                           // data members
  int     n;                       // number of items
  Node*   header;                  // head-of-list sentinel
  Node*   trailer;                 // tail-of-list sentinel
};
```

begin & end 함수



리스트 객체 사용 예시

연산	출력	L
insertFront(8)	–	(8)
$p =$ begin()	p : (8)	(8)
insertBack(5)	–	(8,5)
$q = p;\ ++q$	q : (5)	(8,5)
$p ==$ begin()	*true*	(8,5)
insert(q,3)	–	(8,3,5)
$*q = 7$	–	(8,3,7)
insertFront(9)	–	(9,8,3,7)
eraseBack()	–	(9,8,3)
erase(p)	–	(9,3)
eraseFront()	–	(3)

NodeList 클래스

```
struct Node {          // a node of the list
    Elem elem;           // element value
    Node* prev;          // previous in list
    Node* next;          // next in list
  };

class Iterator {       // an iterator for the list
  public:
    Elem& operator*();         // reference to the element
    bool operator==(const Iterator& p) const;   // compare positions
    bool operator!=(const Iterator& p) const;
    Iterator& operator++();      // (prefix version) move to next position
    Iterator& operator--();      // (prefix version) move to previous position
    friend class NodeList;       // give NodeList access
  private:
    Node* v;             // pointer to the node
    Iterator(Node* u);       // create from node
  };
```

NodeList 클래스

```
NodeList::Iterator::Iterator(Node* u)       { v = u; }

Elem& NodeList::Iterator::operator*()
  { return v->elem; }

bool NodeList::Iterator::operator==(const Iterator& p) const
  { return v == p.v; }

bool NodeList::Iterator::operator!=(const Iterator& p) const
  { return v != p.v; }

NodeList::Iterator& NodeList::Iterator::operator++()
  { v = v->next; return *this; }

NodeList::Iterator& NodeList::Iterator::operator--()
  { v = v->prev; return *this; }
```

NodeList 클래스

```
NodeList::NodeList() {        // constructor
  n = 0;                       // initially empty
  header = new Node;           // create sentinels
  trailer = new Node;
  header->next = trailer;      // have them point to each other
  trailer->prev = header;
}

int NodeList::size() const          // list size
  { return n; }

bool NodeList::empty() const          // is the list empty?
  { return (n == 0); }

NodeList::Iterator NodeList::begin() const
  { return Iterator(header->next); }

NodeList::Iterator NodeList::end() const
  { return Iterator(trailer); }
```

NodeList 클래스

```
void NodeList::insert(const NodeList::Iterator& p, const Elem& e) {
  Node* w = p.v;                  // p's node
  Node* u = w->prev;              // p's predecessor
  Node* v = new Node;             // new node to insert
  v->elem = e;
  v->next = w;  w->prev = v;      // link in v before w
  v->prev = u;  u->next = v;      // link in v after u
  n++;
}

void NodeList::insertFront(const Elem& e) // insert at front
  { insert(begin(), e); }

void NodeList::insertBack(const Elem& e) // insert at rear
  { insert(end(), e); }
```

NodeList 클래스

```
void NodeList::erase(const Iterator& p) { // remove p
  Node* v = p.v;                  // node to remove
  Node* w = v->next;              // successor
  Node* u = v->prev;              // predecessor
  u->next = w;  w->prev = u;      // unlink p
  delete v;                       // delete this node
  n--;                            // one fewer element
}

void NodeList::eraseFront()       // remove first
  { erase(begin()); }

void NodeList::eraseBack()        // remove last
  { erase(--end()); }
```

STL List 클래스

- list(n): n개 원소들의 공간을 갖는 리스트 생성
- size(): 원소의 개수
- empty(): 리스트 L이 비어있으면 참, 그렇지 않으면 거짓
- front(): L의 첫번째 원소의 참조 정보 반환
- back(): L의 마지막 원소의 참조 정보 반환
- push_front(e): 원소 e의 복사본을 L 처음에 삽입
- push_back(e): 원소 e의 복사본을 L 마지막에 삽입
- pop_front(): L 처음 원소를 제거
- pop_back(): L 마지막 원소를 제거
- 기타 등등

부록: STL Containers & Iterators

- STL container 클래스들: 원소의 모음을 저장하는 자료 구조
  - STL sequence container 클래스들: 순차적인 순서로 저장함
    - vector: 벡터 클래스
    - list: 리스트 클래스
    - deque: 양방향 큐 클래스
  - STL associated container 클래스들: 각 원소는 연관된 키값으로 접근함
    - set, multiset: 집합, 다중집합 클래스
    - map, multimap: 맵, 다중맵 클래스
  - 기타 클래스들
    - stack: 스택 클래스
    - queue: 큐 클래스
    - priority_queue: 우선순위 큐 클래스
- STL Iterator 클래스들:
  - 원소의 주소 정보를 숨기면서 **모든 container 클래스들에 대해서 일관성 있는 원소 접근을 가능하게** 함
  - 포인터 변수와 비슷한 역할을 수행함

부록: Iterator 사용 예제

```
int vectorSum1(const vector<int>& V) {
  int sum = 0;
  for (int i = 0; i < V.size(); i++)
    sum += V[i];
  return sum;
}
```

```
int vectorSum2(vector<int> V) {
  typedef vector<int>::iterator Iterator;
  int sum = 0;
  for (Iterator p = V.begin(); p != V.end(); ++p)
    sum += *p;
  return sum;
}
```

```
int vectorSum3(const vector<int>& V) {
  typedef vector<int>::const_iterator ConstIterator;
  int sum = 0;
  for (ConstIterator p = V.begin(); p != V.end(); ++p)
    sum += *p;
  return sum;
}
```

부록: STL Iterator 기반 Container 함수들

- *p: p가 가리키는 원소를 접근
- xxx(p, q): [p, q) 범위의 원소들을 복사하여 xxx 객체 생성
  - xxx는 생성자 함수의 이름, 즉 container 클래스의 이름
- V.assign(p, q): V의 내용을 삭제하고 다른 객체 U의 [p, q) 범위의 원소들을 V의 새 내용으로 저장함
- V.insert(p, e): e를 복사하여 p 위치의 바로 앞에 삽입하고 이후의 원소들은 한 칸씩 오른쪽으로 이동함
- V.erase(p): p 위치에 있는 V의 원소를 삭제하고 이후의 원소들을 한 칸씩 왼쪽으로 이동함
- V.erase(p, q): [p, q) 범위의 원소들을 삭제하고 이후의 원소들을 왼쪽으로 이동하여 공백을 채움
- V.clear(): V의 모든 원소를 삭제함

⇒ 위 함수들은 모든 container 클래스에서 정의되었음 (단, assign 함수는 set, multiset, map, multimap 클래스에서 정의되지 않았음)

- bidirectional iterator (++p, --p): 이전/다음 원소로 이동
- random-access iterator는 vector와 deque에 대해서 정의됨
  - 예시: p에서 3칸 멀리 위치한 원소의 표시 → p + 3

부록: #include <algorithm>

- 다음 함수들은 vector, deque 클래스에서 정의됨
  - sort(p, q): [p, q) 범위의 원소들을 정렬함
  - random_shuffle(p, q): [p, q) 범위의 원소들을 무작위로 섞음

- 다음 함수들은 vector, deque, list 클래스에서 정의됨
  - reverse(p, q): [p, q) 범위의 원소들을 역순으로 배치함
  - find(p, q, e): [p, q) 범위의 원소들 중에서 원소 e를 가리키는 iterator 반환함 (만약 e가 없으면 q를 반환)
  - min_element(p, q): [p, q) 범위의 원소들 중 최소 원소를 가리키는 iterator 반환함
  - max_element(p, q): [p, q) 범위의 원소들 중 최대 원소를 가리키는 iterator 반환함
  - for_each(p, q, f): [p, q) 범위의 모든 원소에 함수 f를 적용함

6.3절 Sequence 클래스

- 개념적으로 ArrayList와 NodeList의 집합체(union) 성격의 STL 클래스

- NodeList의 모든 함수들 뿐만 아니라 ArrayList 성격의 다음 함수들을 제공함
  - atIndex(i): 인덱스 i를 갖는 원소의 위치 반환함
  - indexOf(p): 위치 p에 있는 원소의 인덱스를 반환함

- 데이터베이스 구현 등에 사용됨

Sequence 클래스 구현 (리스트 기반)

```
class NodeSequence : public NodeList {
 public:
  Iterator atIndex(int i) const;      // get position from index
  int indexOf(const Iterator& p) const;// get index from position
 };

 NodeSequence::Iterator NodeSequence::atIndex(int i) const {
  Iterator p = begin();
  for (int j = 0; j < i; j++) ++p;
  return p;
 }

 int NodeSequence::indexOf(const Iterator& p) const {
  Iterator q = begin();
  int j = 0;
  while (q != p) {      // until finding p
    ++q; ++j;         // advance and count hops
  }
  return j;
 }
```

Sequence 클래스의 Array 구현과 Doubly Linked List 구현 비교

Operation	Array	List
size, empty	1	1
atIndex, indexOf, at	1	***n***
begin, end	1	1
*p, ++p, --p	1	1
set(p,e)	1	1
set(i,e)	1	***n***
insert(i,e), erase(i)	***n***	***n***
insertBack, eraseBack	1	1
insertFront, eraseFront	***n***	1
insert(p,e), erase(p)	***n***	1

Index에 의한 bubbleSort

- atIndex 함수는 배열 기반 구현에서는 O(1) 복잡도를 가지나, 리스트 기반 구현에서는 O(n) 복잡도를 가짐

```
void bubbleSort1(Sequence& S) {
  int n = S.size();
  for (int i = 0; i < n; i++) {          // i-th pass
    for (int j = 1; j < n-i; j++) {
      Sequence::Iterator prec = S.atIndex(j-1);   // predecessor
      Sequence::Iterator succ = S.atIndex(j);     // successor
      if (*prec > *succ) {            // swap if out of order
        int tmp = *prec; *prec = *succ; *succ = tmp;
      }
    }
  }
}
```

Iterator에 기반한 bubbleSort

- Iterator 사용시, 아래 코드는 배열 기반 구현이나 리스트 기반 구현에 상관 없이 동일한 복잡도를 가짐

```
void bubbleSort2(Sequence& S) {
  int n = S.size();
  for (int i = 0; i < n; i++) {          // i-th pass
    Sequence::Iterator prec = S.begin();    // predecessor
    for (int j = 1; j < n-i; j++) {
      Sequence::Iterator succ = prec;
      ++succ;                           // successor
      if (*prec > *succ) {              // swap if out of order
        int tmp = *prec; *prec = *succ; *succ = tmp;
      }
      ++prec;                       // advance predecessor
    }
  }
}
```

감사합니다!