제 2 장
배열

추상 데이타 타입과 C++ 클래스

- **C++는 명세와 구현을 구별하고 사용자로 부터 ADT의 구현을 은닉하기 위해 클래스(class) 제공**
- **C++ 클래스의 구성**

  (1) 클래스 이름: (예: *Rectangle*)

  (2) 데이타 멤버: 데이타 (예: *xLow*, *yLow*, *height*, *width*)

  (3) 멤버 함수: 연산의 집합 (예: *GetHeight*(), *GetWidth*())

  (4) 프로그램 접근 레벨: 멤버 및 멤버함수

  - 공용(public), 보호(protected), 전용(private)
  - public: 프로그램 어디에서도 접근
  - private: 같은 클래스 내, friend로 선언된 함수나 클래스에 의해 접근
  - protected: 같은 클래스 내, 서브클래스, friend에 의해서만 접근

C++에서의 데이타 추상화와 캡슐화(1)

- 데이타 멤버
  - 한 클래스의 모든 데이타 멤버를 private(전용) 또는 protected(보호)로 선언
    → 데이타 캡슐화

- 멤버 함수
  - 외부에서 데이타 멤버에 접근할 필요가 있는 경우
  - 외부에서 기동될 것: public으로 선언
  - 나머지: private나 protected로 선언
    (다른 멤버 함수에서 호출)
  - C++에서는 멤버함수의 구현을 클래스 정의 내에 포함 가능 → 인라인(inline) 함수로 처리

C++에서의 데이타 추상화와 캡슐화(2)

- **클래스 연산의 명세와 구현의 분리**
  - 명세 부분에서 함수 프로토타입(함수 이름, 인자 타입, 결과 타입) 정의

    → 헤더 화일

    (예) *Rectangle.h* 파일
  - 함수의 기능 기술: 주석(comment)을 이용
  - 함수의 구현: 같은 이름의 소스 화일에 저장

    (예) *Rectangle.cpp* 파일

클래스 객체의 선언과 멤버 함수의 기동

- 클래스 객체: 변수와 똑같이 선언되고 생성됨
- 객체 멤버들에 대한 접근/기동
  - 점(.) : 직접 구성 요소 선택
  - 화살표(->) : 포인터를 통한 간접 구성 요소 선택

```
// 소스화일 prog.cpp 의 내용
#include <iostream>
#include "Rectangle.h"
int main() { // int 삽입 요망
        Rectangle r, s;          //  r과 s는 Class Rectangle의 객체들이다.
        Rectangle *t = &s;       //  t는 클래스 객체 s에 대한 포인터이다.
        .
        .
// 클래스 객체의 멤버를 접근하기 위해서는 점(.)을 사용한다.
// 포인터를 통해 클래스 객체의 멤버를 접근하기 위해서는 →를 사용한다.
        if (r.GetHeight()*r.GetWidth()  >  t→GetHeight() * t→GetWidth())
            cout << " r ";
        else cout << " s ";
        cout << "has the greater area" << endl;
}
```

어떻게 Rectangle 객체가 선언되고 멤버 함수가 기동되는가를 예시하는 C++코드 부분

생성자(constructor)와 파괴자(destructor)

- 생성자와 파괴자: 클래스의 특수한 멤버 함수
- 생성자
  - 한 객체의 데이타 멤버들을 초기화
  - 클래스 객체가 만들어질 때 자동적으로 실행
  - 해당 클래스의 공용 멤버 함수로 선언
  - 공용 멤버 함수로 선언
  - 생성자 이름은 클래스의 이름과 동일

```
Rectangle::Rectangle(int x, int y, int h, int w)
{
        xLow = x; yLow = y;
        height = h; width = w;
}
```

Rectangle 클래스의 생성자 정의

생성자(constructor)와 파괴자(destructor)

- 생성자를 이용한 **Rectangle** 객체 초기화

```
Rectangle r(1,3,6,6);
Rectangle *s = new Rectangle(0,0,3,4);
Rectangle t;                    // 컴파일 시간 오류! → 기정 생성자 필요
```

- **Rectangle 클래스의 세련된 생성자 정의**

```
Rectangle::Rectangle (int x = 0, int y = 0, int h = 0, int w = 0)
: xLow(x), yLow(y), height(h), width(w)
{ }
```

  - 데이터 멤버들이 멤버 초기화 리스트에 의해 초기화
  - 보다 효율적

생성자(constructor)와 파괴자(destructor)

- **파괴자**
  - 객체가 없어지기 직전 데이타 멤버들을 삭제
  - 클래스 객체가 범위를 벗어나거나 삭제될 때 자동적으로 기동
  - 해당 클래스의 공용 멤버로 선언
  - 이름은 클래스 이름과 동일, 단 앞에 틸데(~) 붙임
  - 반환 타입이나 반환 값을 명기해서도 안되고 인자를 받아서도 안됨
  - 삭제되는 객체의 한 데이타 멤버가 다른 객체에 대한 포인터인 경우
    - 포인터에 할당된 기억장소는 반환
    - 지시되는 객체는 삭제되지 않음

      → 명시적으로 수행하는 파괴자를 정의해야 됨

연산자 다중화(operator overloading)

- **C++는 특정 데이터 타입을 위한 연산자를 구현하는 정의를 허용**
  - 이를 통해 사용자 정의 데이터 타입에 대한 연산자 다중화 허용
  - 클래스 멤버 함수나 보통의 함수 형식
  - 사용될 함수 프로토타입은 특정 연산자의 명세와 일치해야 됨

(예) 연산자 ==

- 두 실수(**float**) 데이타의 동등성 검사
- 두 정수(**int**) 데이타의 동등성 검사
- 두 사각형(Rectangle) 객체 동등성 비교 : 연산자 다중화 필요
  - 같은 객체
  - 위치와 크기가 같은 사각형

연산자 다중화(operator overloading)

- **Class *Rectangle*을 위한 연산자 ==의 다중화**

```
bool Rectangle::operator==(const Rectangle & s ){
    if (this == &s) return true;
    if (xLow == s.xLow) && (yLow == s.yLow)
        && (height == s.height) && (width==s.width) return true;
    else return false;
}
```

- **this:** 멤버 함수를 기동 시킨 특정 클래스 객체에 대한 포인터**(self)**
- ***this:** 클래스 객체 자신
- 비교과정
  - 두 피연산자가 동일 객체이면 결과는 **true** (**this** == &s)
  - 아니면 데이터 멤버를 개별적으로 비교하여 모두 동일하면 결과는 **true**
  - 아니면 결과는 **false**

연산자 다중화(operator overloading)

- **class** *Rectangle* 을 위한 << 연산자의 다중화

```
ostream& operator  << (ostream& os, Rectangle& r)
{
   os << "Position is: “ << r.xLow << " ";
   os << r.yLow < endl;
   os << "Height is: “ << r.height << endl;
   os << "Width is: "r.width << endl;
   return os;
}
```

- 클래스 **Rectangle**의 전용 데이터 멤버에 접근
  → *Rectangle*의 **friend**로 선언되어야 함
  friend *ostream*& **operator**<<(*ostream*&, *Rectangle*&);
- **cout** << *r* 의 출력 예
  - *r*의 왼쪽하단 좌표가 1,3이고 한변이 6인경우)

```
Position is: 1  3
Height is: 6
Width is: 6
```

연산자 다중화(operator overloading)

```
void Print (Rectangle& r)
{
   cout << "Position is: “ << r.xLow << " ";
   cout << r.yLow < endl;
   cout << "Height is: “ << r.height << endl;
   cout << "Width is: "r.width << endl;
}
```

Print(r);

```
void Print (ostream& os, Rectangle& r)
{
   os << "Position is: “ << r.xLow << " ";
   os << r.yLow < endl;
   os << "Height is: “ << r.height << endl;
   os << "Width is: "r.width << endl;
}
```

Print(cout, r);

```
ostream& operator << (ostream& os, Rectangle& r)
{
   os << "Position is: “ << r.xLow << " ";
   os << r.yLow < endl;
   os << "Height is: “ << r.height << endl;
   os << "Width is: "r.width << endl;
   return os;
}
```

operator<<(cout, r);

cout << r;

기타 내용

- C++의 **struct**
  - 묵시적 접근 레벨 = **public**
    (cf. **class**에서는 **private**)
  - C의 **struct**에 대한 일반화
    (멤버 함수도 정의 가능)

- **union**
  - 제일 큰 멤버에 맞는 저장장소 할당
  - 보다 효율적으로 기억장소 사용

- **static(정적) 클래스 데이타 멤버**
  - 클래스에 대한 전역 변수: 오직 하나의 사본
  - 모든 클래스 객체는 이를 공유
  - 정적 데이타 멤버는 클래스 외부에서 정의

ADT와 C++ 클래스

- **ADT(Abstract Data Type, 추상 데이터 타입)**
  - An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects.
- **C++의 class**
  - A representation of ADT(Abstract Data Type)
  - 묵시적 접근 레벨 = **private**
    (cf. **struct**에서는 **private**)
  - C의 **struct**에 대한 일반화
    (멤버 함수도 정의 가능)

추상 데이타 타입으로서의 배열(The Array ADT)

- 일련의 연속적인 메모리 위치: 구현 중심
- <*인덱스,값*>: <***index,value***>의 쌍 집합
  - *index* -> *value* : 대응(correspondence) 또는 사상(mapping)

  *value*는 **float**로 가정

  *index*는 *indexset* 의 member

  1차원인 경우 *indexset*은 가령 {0, …, *n*-1}

  2차원인 경우 *indexset*은 가령 {(0,0), (0,1), (0,2),(1,0),(1,1),(1,2), (2,0),(2,1),(2,2)}
- **C++ 배열**
  - 인덱스 집합이 0부터 시작
  - 배열의 인덱스 범위를 확인하지 않음

  (예) **float** *floatArray*[*n*];

  *i*번째 원소값 : *floatArray*[*i*], *(*floatArray*+*i*)
- 함수
  - *GeneralArray*(**int** *j*, *RangeList list*, **float** *initValue=defaultValue*)
    - Creates *j*-dimensional array of floats; the range of *k*-th dimension is given by the *k*-th element of list. For each index in the index set, insert <*i*., *initValue*> into the array.
  - *Retrieve*(*index*)
  - *Store*(*index*, *item*)

다항식 추상 데이타 타입(The Polynomial ADT)

- **순서 리스트, 선형 리스트**
  - examples
    - 한 주일의 요일: (일,월,화,수, .., 토)
    - 카드 한 벌의 값: (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King)
    - 건물 층: (지하실, 로비, 일층, 이층)
    - 미국의 제2차 세계대전 참전년도: (1941, 1942, 1943, 1944, 1945)
    - 스위스의 제2차 세계대전 참전년도: ()
  - 리스트 형태 : $(a_0, a_1, \ldots, a_{n-1})$
  - 공백 리스트의 예 : ()

순서 리스트, 선형 리스트

- 리스트에 대한 연산
  - 리스트의 길이 n의 계산
  - 리스트의 항목을 왼쪽에서 오른쪽(오른쪽에서 왼쪽)으로 읽기
  - 리스트로부터 i번째 항목을 검색, $0 \le i < n$
  - 리스트의 i번째 항목을 대체, $0 \le i < n$
  - 리스트의 i번째 위치에 새 항목 삽입, $0 \le i < n$ 이것은 원래 $i, i+1, \ldots, n-1$ 항목 번호를 $i+1, i+2, \ldots, n$으로 만듦
  - 리스트의 i번째 항목을 제거, $0 \le i < n$ 이것은 원래 $i+1, \ldots, n-1$ 항목 번호를 $i, i+1, \ldots, n-2$로 만듦
- 순서 리스트의 일반적인 구현
  - 배열을 이용

    인덱스 $i \rightarrow a[i], 0 \le i < n$

    ... i, $i+1$, ...

    ... $a[i]$, $a[i+1]$, ...
  - 순차 사상(sequential mapping)
  - 문제점: 삽입, 삭제시 오버헤드

다항식 (polynomial)

- $a(x)=3x^2+2x-4$, $b(x)=x^8-10x^5-3x^3+1$
  - 계수(coefficient) : 3, 2, -4
  - 지수(exponent) : 2, 1, 0
  - 변수(variable) : x
- 차수(degree): 0이 아닌 제일 큰 지수
- 다항식의 합과 곱
  - $a(x) + b(x) = \sum(a_i + b_i)x^i$
  - $a(x) \cdot b(x) = \sum(a_i x^i \cdot \sum(b_j x^j))$

다항식 표현

- **첫번째 결정 : 서로 다른 지수들은 내림차순으로 정돈**
- **[표현 1] Polynomial의 전용 데이타 멤버 선언**

```
private:
        int degree;                  //degree ≤ MaxDegree
        float coef [MaxDegree + 1];  // 계수 배열
```

  - *a*가 Polynomial 클래스 객체, $n \le MaxDegree$

$a.degree = n$
$a.coef[i] = a_{n-i}, \quad 0 \le i \le n$

  - *a.coef*[*i*]는 x^{n-i}의 계수, 각 계수는 지수의 내림차순으로 저장
  - 대부분 다항식의 연산(덧셈, 뺄셈, 계산, 곱셈 등)을 위한 알고리즘을 간단하게 구성

다항식 표현

- **[표현 2] Polynomial의 전용 데이타 멤버 선언**

```
private:
        int degree;
        float *coef;
```

  - 생성자를 Polynomial에 추가

```
Polynomial::Polynomial(int d){
        degree=d;
        coef=new float[degree+1];
}
```

  - 희소 다항식에서 기억 공간 낭비
    (예) 다항식 $x^{1000}+1$
    → *coef*에서 999개의 엔트리는 0
    → 표현 3방법(클래스 term) 이용

다항식 표현

- **[표현 3] 모든 다항식은 배열 termArray를 이용해 표현**
  - termArray의 각 원소는 term 타입
  - Polynomial의 정적 클래스 데이터 멤버

```
class Polynomial;    //전방 선언
class Term{
friend class Polynomial; // class명시필요(교재에 빠져있음)
friend ostream& operator<<(ostream&, Polynomial&);
friend istream& operator>>(istream&, Polynomial&);
private:
        float coef;   // 계수
        int exp;      // 지수
};
```

```
// simple version by kcl
struct Term{
    float coef;   // 계수
    int exp;      // 지수
};
```

- **Polynomial의 전용 데이타 멤버 선언**

```
private:
    Term *termArray;   // 0이 아닌 항의 배열
     int capacity;     // termArray의 크기
     int terms;        // 0이 아닌 항의 수
```

다항식 덧셈

- $c=a+b$를 구하는 C++ 함수

- 함수 *Add*: $a(x)$(즉 ***this**)와 $b(x)$를 항별로 더하여 $c(x)$를 만드는 함수
  - *Polynomical*의 기정 생성자가 *capacity*와 *terms*를 각각 1과 0으로 초기화하고 *termArray*를 *capacity*개의 *Term*으로 초기화하는 것을 가정

- 기본루프는 지수를 비교한 결과에 따라 두 다항식의 항들을 하나로 합하는 과정으로 구성

- 두 지수(exp)의 크기를 비교하여 알맞은 동작 수행

다항식 (polynomial)

$a(x)=3x^2+2x-4$ $b(x)=x^8-10x^5-3x^3+1$

a

capacity	4
terms	3
termArray	

	coef	*exp*
termArray[0]	3	2
termArray[1]	2	1
termArray[2]	-4	0
termArray[3]	-	-

b

capacity	4
terms	4
termArray	

	coef	*exp*
termArray[0]	1	8
termArray[1]	-10	5
termArray[2]	-3	3
termArray[3]	1	0

c = a+b

capacity	8
terms	6
termArray	

	coef	*exp*
termArray[0]	1	8
termArray[1]	-10	5
termArray[2]	-3	3
termArray[3]	3	2
termArray[4]	2	1
termArray[5]	-3	0
termArray[6]	-	-
termArray[7]	-	-

Adding two polynomials

```
Polynomial Polynomial::Add(Polynomial b)
// return the sum of the polynomials *this and b.
  Polynomial c; // 합을 저장
  int aPos = 0, bPos = 0;
  while ((aPos < terms) && (bPos < b.terms))
    if (termArray[aPos].exp == b.termArray[bPos].exp) {
      float t = termArray[aPos].coef + b.termArray[bPos].coef;
      if (t) c.NewTerm(t, termArray[aPos].exp);
      aPos++; bPos++;
    } else if (termArray[aPos].exp < b.termArray[bPos].exp) {
      c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
      bPos++;
    }else {
      c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
      aPos++;
    }
  // add in remaining terms of *this
  for (;aPos < terms; aPos++)
    c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
  // add in remaining terms of b(x)
  for (;bPos < b.terms ; bPos++) // 교재에는 b++로 잘못 적혀있음
    c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
  return c;
}
```

```
void Polynomial::NewTerm(const float theCoeff, const int theExp)
// 에 새로운 항을 termArray 끝에 첨가
   if (terms==capacity)
   {//termArray의 크기를 두 배로 확장
       capacity *= 2;
       Term *temp = new Term [capacity];     // 새로운 배열
       copy(termArray, termArray + terms, temp);
       delete [ ] termArray;                 // 그전 메모리를 반환
       termArray = temp; }
    termArray[terms].coef = theCoef;
    termArray[terms++].exp = theExp;}
```

새로운 항의 추가와 배열 크기를 두 배 확장

- ***Add*** **의 분석 :**
  - *a*와 *b*에서의 0이 아닌 항의 수로 분석
  - 전체 실행 시간: $O(m+n)$
  - 배열을 두배 늘리는 것은 *Add*의 전체 실행 시간에 대해 O(m+n) 만큼의 시간 복잡도를 가진다. 즉 배열 두배 확장은 *Add* 전체 실행 시간의 아주 작은 부분이 된다는 것을 보여준다.

Array representation of two polynomials

- 두개의 다항식 $A(x)=2x^{1000}+1$, $B(x)=x^4+10x^3+3x^2+1$ 표현

	a.start	*a.finish*	*b.start*			*b.finish*	*free*
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

두 다항식의 배열 표현

- *n*개의 0이 아닌 항을 가진 다항식 *a*는 *a.finish* =*a.start*+*n*-1의 식을 만족하는 *a.start*와 *a.finish*를 가짐
  - *A*가 0이 아닌 항이 없을 경우
    → *a.finish* = *a.start*-1

희소 행렬(Sparse matrices)

- $a[m][n]$
  - $m \times n$ 행렬 a
    - m : 행의 수
    - n : 열의 수
    - $m \times n$: 원소의 수
  - 희소 행렬(sparse matrix)
    - 0이 아닌 원소수 / 전체 원소수 << 1.0
    - → 0이 아닌 원소만 저장할 필요 있으므로 시간/공간 절약
  - 행렬에 대한 연산
    - Creation(생성)
    - Transpose(전치)
    - Addition(덧셈)
    - Multiplication(곱셈)

Two Matrices

	0	1	2
0	−27	3	4
1	6	82	−2
2	109	−64	11
3	12	8	9
4	48	27	47

(a)

	0	1	2	3	4	5
0	15	0	0	22	0	−15
1	0	11	3	0	0	0
2	0	0	0	−6	0	0
3	0	0	0	0	0	0
4	91	0	0	0	0	0
5	0	0	28	0	0	0

(b)

두 행렬

효율적인 희소 행렬 표현

- **표현 방법**
  - 각 항(*MatrixTerm*)은 <*row*, *col*, *val*> 3원소 쌍(triple)임
  - *termArray* 라는 *Term*배열에 *row*순서대로 저장하고 동일 *row*경우 *col*순서로 저장 쌍들은 행 순서로 저장
  - 연산 종료 확인 등에 사용하고자, 행렬의 행의 수(*rows*)와 열의 수(*cols*)와 항의 수(*terms*)를 저장.

- *MatrixTerm*의 C++ 클래스 표현

```
class SparseMatrix;                    // 전방 선언
class MatrixTerm {
friend class SparseMatrix;
friend ostream& operator<<(ostream&, SparseMatrix&);
friend istream& operator>>(istream&, Sparsematrix&);
private:
        int row, col, value;
 };
```

```
// simple version by kcl
struct MatrixTerm {
        int row, col, value;
 };
```

효율적인 희소 행렬 표현

- 클래스 **SparseMatrix** 내부 정의

```
private:
        int rows, cols, terms, capacity;
        MatrixTerm *smArray;
 friend ostream& operator<<(ostream&, SparseMatrix&);
 friend istream& operator>>(istream&, Sparsematrix&);
```

- *rows* : 행의 수
- *cols* : 열의 수
- *terms*: 0이 아닌 항의 총 수
- *capacity*: *smArray*의 확보된 크기

원소 쌍으로 저장된 희소행렬과 전치행렬

Note : *rows* = 6, *cols* = 6, *terms* = 8, *capacity* = 8(or bigger)

	row	*col*	*val*
smArray[0]	0	0	15
[1]	0	3	22
[2]	0	5	-15
[3]	1	1	11
[4]	1	2	3
[5]	2	3	-6
[6]	4	0	91
[7]	5	2	28

(a)

	row	*col*	*val*
smArray[0]	0	0	15
[1]	0	4	91
[2]	1	1	11
[3]	2	1	3
[4]	2	5	28
[5]	3	0	22
[6]	3	2	-6
[7]	5	0	-15

(b)

행렬의 전치(Transpose)

- 원래의 행렬 각 행 *i*에 대해서 원소 (*i*, *j*, 값) 을 가져와서 전치행렬의 원소 (*j*, *i*, 값) 으로 저장

  (예)

  (0, 0, 15) → (0, 0, 15)
  (0, 3, 22) → (3, 0, 22)
  (0, 5, -15) → (5, 0, -15)
  (1, 1, 11) → (1, 1, 11)

  – 올바른 순서 유지위해 기존원소 이동시켜야 하는 경우 발생

- 알고리즘

  for (열 *j*에 있는 모든 원소에 대해)
  원소(*i*, *j*, 값)을 원소(*j*, *i*, 값)으로 저장

행렬의 전치

- **단순 2차원 배열 표현되는 경우**
  - $O(rows \cdot cols)$시간내의 $rows \cdot cols$크기 행렬의 전치를 얻을 수 있음.
  - 가장 단순한 형태의 알고리즘

```
for (int j = 0; j < columns; j++)
    for (int i = 0; i < rows; i++)
        b[j][i] = a[i][j];
```

- *Transpose*[Program 2.10]의 분석
  - 총 실행시간: $O(terms \cdot cols)$
  - ***this**와 *b*가 필요로 하는 공간 외에 이 함수는 변수 *c*, *i*, *currentB*를 위한 고정된 공간만을 추가로 필요로 한다.

행렬의 전치(2)

```
1   SparseMatrix SparseMatrix::Transpose()
2   {// *this의 전치 행렬을 반환한다.
3      SparseMatrix b(cols, rows, terms); // b.smArray의 크기는 terms이다
4      if (terms > 0)
5      {// 0이 아닌 행렬
6         int currentB = 0;\
7         for (int c = 0; c < cols; c++) // 열별로 전치
8            for (int i = 0; i < terms; i++)
9            // 열 c로부터 원소를 찾아 이동시킨다
10               if (smArray[i].col == c)
11               {
12                  b.smArray[currentB].row = c;
13                  b.smArray[currentB].col = smArray[i].row;
14                  b.smArray[currentB++].value = smArray[i].value;
15                }
16     } // if (terms > 0)의 끝
17     return b;
18  }
```

행렬의 전치(3)

- 메모리를 조금 더 사용한 개선 알고리즘 : *FastTranspose*
  - 먼저 행렬 ***this**의 각 열에 대한 원소 수를 구함
  - 전치 행렬 *b*의 각 행의 원소 수(즉 행렬 *a*의 각 열의 원소수)를 결정
  - 이 정보에서 전치행렬 *b*의 각행의 시작위치 구함
  - 원래 행렬 *a*에 있는 원소를 하나씩 전치 행렬 *b*의 올바른 위치로 옮김
  - 실행시간: $O(cols+terms)$

	[0]	[1]	[2]	[3]	[4]	[5]		
rowSize =	2	1	2	2	0	1	←	# of terms in *b*'s row(*a*'s col)
rowStart =	0	2	3	5	7	7	←	starting position of *b*'s row

(!!교재수정 요망)

행렬의 전치(4)

```
1   SparseMatrix SparseMatrix::FastTranspose()
2   {// *this의 전치를 O(terms + cols) 시간에 반환한다.
3      SparseMatrix b(cols, rows, terms);
4      if (terms > 0)
5      {// 0이 아닌 행렬
6         int *rowSize = new int[cols];
7         int *rowStart = new int[cols];
8         // compute rowSize[i] = b의 행 i에 있는 항의 수를 계산
9         fill(rowSize, rowSize + cols, 0); // 초기화
10        for (i = 0; i < terms; i++) rowSize[smArray[i].col]++;
11        // rowStart[i] = b의 행 i의 시작점
12        rowStart = 0;
13        for (i = 1; i < cols; i++) rowStart[i] = rowStart[i-1] + rowSize[i-1];
14        for (i = 0; i < terms; i++)
15        {// *this를 b로 복사
16           int j = rowStart[smArray[i].col];
17           b.smArray[j].row = smArray[i].col;
18           b.smArray[j].col = smArray[i].row;
19           b.smArray[j].value = smArray[i].value;
20           rowStart[smArray[i].col]++;
21        } // for의 끝
22        delete [ ] rowSize;
23        delete [ ] rowStart;
24     } // if의 끝
25     return b;
26  }
```

행렬 곱셈

- $a(m \times n)$와 $\times\ b(n \times p)$의 곱셈
- d차원은 $m \times p$

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} \qquad 0 \le i < m,\ 0 \le j < p$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

- 순서 리스트로 표현된 두 희소 행렬의 곱셈
  - *result*의 원소를 행별로 계산
  - 이전 계산 원소를 이동하지 않고 적절한 위치에 저장
  - 행렬 a의 한 행을 선택하고 $j = 0,1,...,b.cols-1$에 대해 b의 j열에 있는 모든 원소를 찾음
  - b를 전치
  - a의 i행과 b의 j열의 원소들이 정해지면 다항식 덧셈과 유사한 합병연산 수행

희소 행렬의 곱셈

- 매우 비효율적인 곱셈 알고리즘 – 시작 복잡도가 크다

```
for (int i = 0; i < a.rows; i++)
    for (int j = 0; j < b.cols; j++)
   {
          sum = 0;
          for (int k = 0; k < a.cols; k++)
                sum += (a[i][k] * b[k][j]);
          c[i][j] = sum;
    }
```

배열의 표현

- 배열이 $a[u_1][u_2]$, ….., $[u_n]$일 경우
- a의 총 원소수: $\prod_{i=1}^{n} u_i$
- 표현 순서: 행우선(row major order), 열우선(column major order)
  - 행우선(row major order)
    - 총 원소수: 2*3*2*2=24
    - 저장 순서:

      $a[0][0][0][0]$, $a[0][0][0][1]$, $a[0][0][1][0]$, $a[0][0][1][1]$
      $a[0][1][0][0]$, $a[0][1][0][1]$, $a[0][1][1][0]$, $a[0][1][1][1]$

      ……

      $a[1][2][0][0]$, $a[1][2][0][1]$, $a[1][2][1][0]$, $a[1][2][1][1]$
    - 즉, 0000, 0001, …, 1210, 1211

      -> 사전순서(lexicographic order)

배열의 표현

- 배열 원소 $a[i_1][i_2],\ldots,[i_n]$의 1차원 배열 위치로의 변환
  (예) $a[0][0][0][0]$ ⇒ 위치 0
  $a[0][0][0][1]$ ⇒ 위치 1
  $a[1][2][1][1]$ ⇒ 위치 23

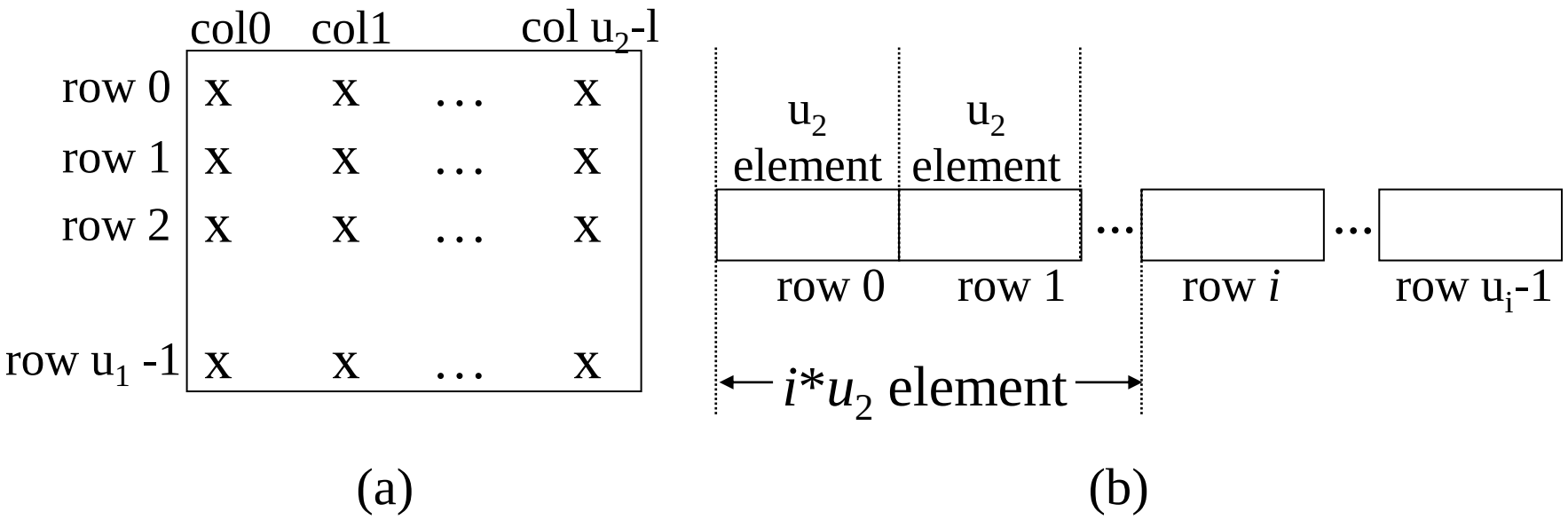
- 1차원 배열 $a[u_1]$
  - α: $a[0]$의 주소
  - 임의의 원소 $a[i]$의 주소 : α +i

배열 원소	$a[0]$	$a[1]$	...	$a[i]$	...	$a[u_1\text{-}1]$
주소	α	α+1	...	α+i	...	α+u_1-1

$a[u_1]$의 순차적인 표현

배열의 표현

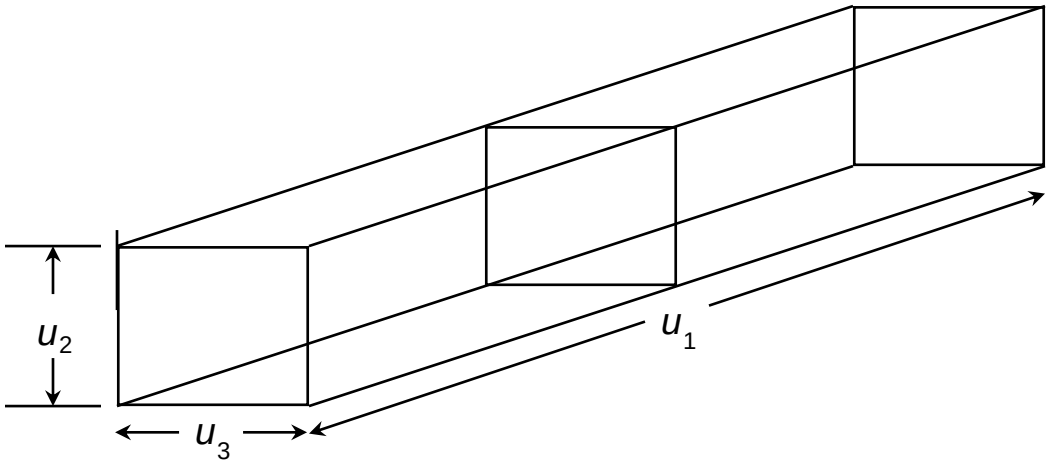
- **2차원 배열**
  - α :*a*[0][0]의 주소
  - *a*[u_1][u_2]
  - *a*[*i*][0]의 주소: α +*i**u_2
  - *a*[*i*][*j*]의 주소: α+*i**u_2+*j*



a[u_1][u_2]의 순차적 표현

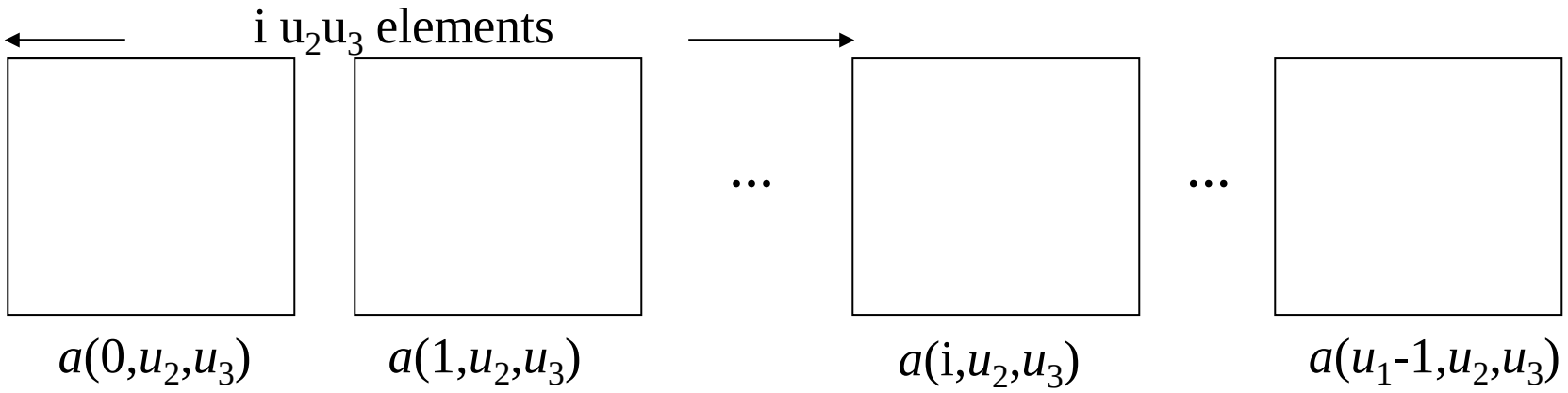
배열의 표현

- **3차원 배열**
  - $a[u_1][u_2][u_3]$
  - α ;$a[0][0][0]$의 주소
  - $a[i][0][0]$의 주소: $\alpha + iu_2u_3$
  - $a[i][j][k]$의 주소: $\alpha + iu_2u_3 + ju_3 + k$



(a) 3차원 배열 $a[u_1][u_2][u_3]$ 가 u_1개의 2차원 배열로 취급됨

다차원 배열의 표현



(b) 3차원 배열의 순차 행 우선 표현

- n차원 배열 $a[u_1][u_2]\ldots,[u_n]$
  - α ; $a[0][0],\ldots,[0]$의 주소
  - $a[i_1][0][0],\ldots,[0]$의 주소: $\alpha + i_1u_2u_3 \ldots u_n$
  - $a[i_1][i_2][0],\ldots,[0]$의 주소: $\alpha + i_1u_2u_3 \ldots u_n + i_2u_3 \ldots u_n$
  - $a[i_1][i_2],\ldots,[i_n]$의 주소

$$\alpha + \sum_{j=1}^{n} i_j a_j \qquad \text{여기서} \quad \begin{cases} a_j = \prod_{k=j+1}^{n} u_k, & 1 \le j < n \\ a_n = 1 \end{cases}$$

스트링 추상 데이타 타입

- 문자열(string): $S = s_0, ..., s_{n-1}$의 형태,
  - s_i : 문자 집합의 원소
  - $n = 0$: 공백 또는 널 문자열

- 연산
  - 새로운 공백 스트링 생성, 스트링 읽기 또는 출력,
  - 두 스트링 접합(concatenation), 스트링 복사,
  - 스트링 비교, 서브스트링을 스트링에 삽입,
  - 스트링에서 서브스트링 삭제,
  - 스트링에서 특정 패턴 검색

스트링 패턴 매치: 간단한 알고리즘

- 함수 *Find*
  - 두 개의 스트링 *s*와 *pat*
  - *pat*이 *s*에서 탐색할 패턴
  - 호출형식: *s.Find*(*pat*)
  - *pat*과 *i*번째 위치에서 시작하는 *s*의 부분문자열
  - 부합될 때 인덱스 *i*를 반환
  - *pat*이 공백이거나 s의 부분문자열이 아닌 경우 -1을 반환
  - *LengthP*: 패턴 *pat*의 길이
  - *LengthS*: 스트링 *s*의 길이
  - *s*에서 위치 *LengthS-LengthP*의 오른쪽은 *pat*과 매치될 문자가 충분하지 않으므로 고려하지 않아도 됨
  - 복잡도: *O*(*LengthP*▪ *LengthS*)