

제 2 장

배열

추상 데이터 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) 데이터 멤버: 클래스를 구성하는 데이터 (예: *xLow*, *yLow*, *height*, *width*)
 - (3) 멤버 함수: 연산의 집합 (예: *GetHeight()*, *GetWidth()*)
 - (4) 프로그램 접근 레벨: 멤버 및 멤버함수
 - ◆ public: 프로그램 어디에서도 접근
 - ◆ private: 같은 클래스 내, friend로 선언된 함수나 클래스에 의해 접근
 - ◆ protected: 같은 클래스 내, 서브클래스, friend에 의해서만 접근

C++에서의 데이터 추상화와 캡슐화(1)

◆ 데이터 멤버

- 한 클래스의 모든 데이터 멤버를 `private`(전용) 또는 `protected`(보호)로 선언
- `public`(공용) 멤버 함수로 접근

◆ 멤버 함수

- 외부에서 데이터 멤버에 접근할 필요가 있는 경우 `public`으로 선언
- 나머지: `private`나 `protected`로 선언
(다른 멤버 함수에서 호출)
- C++에서는 멤버함수의 구현을 클래스 정의 내에 포함 가능 → 인라인(`inline`) 함수로 처리

C++에서의 데이터 추상화와 캡슐화(2)

◆ 클래스 연산의 명세와 구현의 분리

- 명세 부분에 함수 프로토타입(함수 이름, 인자 타입, 결과 타입)

→ 헤더 파일

(예) *Rectangle.h* 파일

- 함수의 기능 기술: 주석(comment)을 이용
- 함수의 구현: 같은 이름의 소스 파일에 저장

(예) *Rectangle.cpp* 파일

C++에서의 데이터 추상화와 캡슐화(2)

Program 2.1: Definition of the C++ class Rectangle

```
=====
#ifndef RECTANGLE_H
#define RECTANGLE_H
// In the header file Rectangle.h
class Rectangle {
public:          // the following members are public
    // the next four members are member functions
    Rectangle();      // constructor
    ~Rectangle();     // destructor
    int GetHeight();  // returns the height of the rectangle
    int GetWidth();   // returns the width of the rectangle
private:        // the following members are private
    // the following members are data members
    int xLow, yLow, height, width;
    // (xLow, yLow) are the coordinates of the bottom left corner of the rectangle
};
#endif
=====
```

C++에서의 데이터 추상화와 캡슐화(2)

Program 2.2: Implementation of operations on Rectangle

=====

// In the source file Rectangle.cpp

#include "Rectangle.h"

// The prefix "Rectangle::" identifies GetHeight() and GetWidth() as member
// functions belonging to class Rectangle. It is required because the member
// functions are implemented outside their class definition

int Rectangle::GetHeight() { return height; }

int Rectangle::GetWidth() { return width; }

=====

클래스 객체의 선언과 멤버 함수의 기동

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

```
// 소스파일 main.cpp 속에
#include <iostream>
#include "Rectangle.h"
main() {
    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;
}
```

생성자(constructor)와 파괴자(destructor)

- ◆ 생성자와 파괴자: 클래스의 특수한 멤버 함수
- ◆ 생성자
 - 한 객체의 데이터 멤버들을 초기화
 - 클래스 객체가 만들어질 때 자동적으로 실행
 - 해당 클래스의 public 멤버 함수로 선언
 - 생성자 이름은 클래스의 이름과 동일
 - 리턴형을 명시하지 않으며 값을 리턴하지도 않음

```
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! → 디폴트 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)  
{ }
```

생성자(constructor)와 파괴자(destructor)

◆ 파괴자

- 클래스 객체가 범위를 벗어나거나 삭제될 때 자동적으로 기동
- 해당 클래스의 공용 멤버로 선언
- 이름은 클래스 이름과 동일, 단 앞에 틸데(~) 붙임
- 리턴형을 명시하지 않으며 값을 리턴하지도 않으며 인자를 받아서도 안됨
- 삭제되는 객체의 한 데이터 멤버가 다른 객체에 대한 포인터인 경우
 - ◆ 포인터에 할당된 기억장소는 반환
 - ◆ 포인터가 가리키는 객체는 삭제되지 않음
 - 명시적으로 포인터가 가리키는 객체를 해제하는 파괴자를 정의해야 됨

연산자 다중화(operator overloading)

- ◆ 사용자 정의 데이터 타입에 대한 연산자 다중화 허용
 - 클래스 멤버 함수 혹은 보통의 함수 형식으로 다중화

(예) 연산자 ==

- ◆ 두 실수(float) 데이터의 동등성 검사
- ◆ 두 정수(int) 데이터의 동등성 검사
- ◆ 두 사각형(Rectangle) 객체 동등성 비교 : 연산자 다중화 필요

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 **private** 데이터 멤버에 접근
→ **Rectangle**의 friend로 만들어져야 됨
- ◆ **cout** << *r* 의 출력 예
Position is: 1 3
Height is: 6
Width is: 6

기타 내용

◆ C++의 struct

- 묵시적 접근 레벨 = public
(cf. class에서는 private)
- 멤버 함수도 정의 가능

◆ union

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

◆ static(정적) 클래스 데이터 멤버

- 클래스에 대한 전역 변수: 오직 하나의 사본
- 모든 클래스 객체는 이를 공유
- 정적 데이터 멤버는 클래스 외부에서 정의

다항식 추상 데이터 타입

◆ 순서 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 : $(a_0, a_1, \dots, a_{n-1})$

– 공백 리스트의 예 : $()$

순서 리스트

- 리스트에 대한 연산

- ◆ 리스트의 길이 n 의 계산
- ◆ 리스트의 항목을 왼쪽에서 오른쪽(오른쪽에서 왼쪽)으로 읽기
- ◆ 리스트로부터 i 번째 항목을 검색, $0 \leq i < n$
- ◆ 리스트의 i 번째 항목을 대체, $0 \leq i < n$
- ◆ 리스트의 i 번째 위치에 새 항목 삽입, $0 \leq i < n$ 이것은 원래 $i, i+1, \dots, n-1$ 항목 번호를 $i+1, i+2, \dots, n$ 으로 만들
- ◆ 리스트의 i 번째 항목을 제거, $0 \leq i < n$ 이것은 원래 $i+1, \dots, n-1$ 항목 번호를 $i, i+1, \dots, n-2$ 로 만들

- 순서 리스트의 일반적인 구현

- ◆ 배열을 이용
- ◆ 문제점: 삽입, 삭제 시 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
 - 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 (polynomial)

ADT 2,3: Abstract data type Polynomial

=====

```
class Polynomial {  
    //  $p(x) = a_0 x^{e_0} + \dots + a_n x^{e_n}$ ; a set of ordered pairs  
    // of  $\langle e_i, a_i \rangle$ ,  
    // where  $a_i$  is a nonzero float coefficient and  $e_i$  is a non-negative integer  
    // exponent.  
public:  
    Polynomial();  
    // Construct the polynomial  $p(x) = 0$ .  
  
    Polynomial Add(Polynomial poly);  
    // Return the sum of the polynomials *this and poly.  
  
    Polynomial Mult(Polynomial poly);  
    // Return the product of the polynomials *this and poly.  
  
    float Eval(float f);  
    // Evaluate the polynomial *this at f and return the result.  
};
```

=====

다항식 표현

- ◆ 첫번째 결정 : 다항식을 지수 내림차순으로 정돈
- ◆ [표현 1]

private:

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

- a가 Polynomial 클래스 객체, $n \leq \text{MaxDegree}$

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

- a.coef[i]는 x^{n-i} 의 계수, 각 계수는 지수의 내림차순으로 저장
- 다항식의 연산을 위한 알고리즘이 간단하지만 메모리 낭비가 심함

다항식 표현

◆ [표현 2]

```
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 $x^{1000}+1$
→ coef에서 999개의 엔트리는 0

다항식 표현

◆ [표현 3]

- termArray의 각 원소는 term 타입

```
class Polynomial; //전방 선언
class term{
friend Polynomial;
private:
    float coef; // 계수
    int exp;    // 지수 };
```

◆ Polynomial의 전용 데이터 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: $a(x)(*this)$ 와 $b(x)$ 를 항별로 더하여 $c(x)$ 를 만드는 함수
 - Polynomial의 디폴트 생성자가 capacity와 terms를 각각 **1**과 **0**으로 초기화하고 termArray를 1로 초기화하는 것을 가정
- ◆ 기본루프는 지수를 비교한 결과에 따라 두 다항식의 항들을 하나로 합하는 과정으로 구성

```

1 Polynomial Polynomial::Add(Polynomial b)
2{// Return the sum of of the polynomials *this and b.
3   Polynomial c;
4   int aPos = 0, bPos = 0;
5   while ((aPos < terms) && (bPos < b.terms))
6       if ((termArray[aPos].exp == b.termArray[bPos].exp) {
7           float t = termArray[aPos].coef + b.termArray[bPos].coef;
8           if (t) c.NewTerm(t, termArray[aPos].exp);
9           aPos++; bPos++;
10      }
11      else if ((termArray[aPos].exp < b.termArray[bPos].exp) {
12          c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
13          bPos++;
14      }
15      else {
16          c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
17          aPos++;
18      }
19  // add in remaining terms of *this
20  for (; aPos < terms ; aPos++)
21      c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
22  // add in remaining terms of b(x)
23  for (; bPos < b.terms; b++)
24      c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
25  return c;
26}

```

```

void Polynomial::NewTerm(const float the Coeff, const int theExp)
// 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 :

- 전체 실행 시간: $O(m+n)$

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 \ll small
 - ◆ \rightarrow 0이 아닌 원소만 저장한다면 시간과 공간 절약
- 행렬에 대한 연산
 - ◆ Creation(생성)
 - ◆ Transpose(전치)
 - ◆ Addition(덧셈)
 - ◆ Multiplication(곱셈)

밀집 행렬과 희소 행렬 예

$$\begin{array}{c}
 \begin{array}{ccc}
 & 0 & 1 & 2 \\
 0 & \begin{bmatrix} -27 & 3 & 4 \end{bmatrix} \\
 1 & 6 & 82 & -2 \\
 2 & 109 & -64 & 11 \\
 3 & 12 & 8 & 9 \\
 4 & 48 & 27 & 47
 \end{array}
 \end{array}$$

(a)

$$\begin{array}{c}
 \begin{array}{cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
 0 & \begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \end{bmatrix} \\
 1 & 0 & 11 & 3 & 0 & 0 & 0 \\
 2 & 0 & 0 & 0 & -6 & 0 & 0 \\
 3 & 0 & 0 & 0 & 0 & 0 & 0 \\
 4 & 91 & 0 & 0 & 0 & 0 & 0 \\
 5 & 0 & 0 & 28 & 0 & 0 & 0
 \end{array}
 \end{array}$$

(b)

두 행렬

효율적인 희소 행렬 표현

◆ 표현 방법

- <행, 열, 값> 3원소 쌍(triple)으로 유일하게 식별 가능

```
class SparseMatrix;      // 전방 선언
class MatrixTerm {
friend class SparseMatrix
private:
    int row, col, value; // 행 번호, 열 번호, 값
};
```

```
private:
    int rows, cols, terms, capacity; // 행, 열, 0이 아닌 항의 총수, 배열크기
    MatrixTerm *smArray;
```

효율적인 희소 행렬 표현

◆ 표현 방법

	0	1	2	3	4	5
0	15	0	0	22	0	-15
1	0	11	3	0	22	-15
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

		row	col	value
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

원소 쌍으로 저장된 희소행렬과 전치행렬

		행	렬	값
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

		행	렬	값
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

행렬의 전치

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

(예)

for (각 i 행에 대해)

원소 $(i, j, 값)$ 을 원소 $(j, i, 값)$ 으로 저장

$(0, 0, 15) \rightarrow (0, 0, 15)$

$(0, 3, 22) \rightarrow (3, 0, 22)$

$(0, 5, -15) \rightarrow (5, 0, -15)$

$(1, 1, 11) \rightarrow (1, 1, 11)$

- 올바른 순서 유지 위해 기존 원소를 이동시켜야 하는 경우 발생

- ◆ 개선된 알고리즘

for (열 j 에 있는 모든 원소에 대해)

원소 $(i, j, 값)$ 을 원소 $(j, i, 값)$ 으로 저장

행렬의 전치

Program 2.10: Transposing a matrix

=====

```
1 SparseMatrix SparseMatrix::Transpose( )
2{// Return the transpose of *this.
3 SparseMatrix b(cols, rows, terms); // capacity of b, smArray is terms
4 if (terms > 0)
5     { // nonzero matrix
6         int currentB = 0 ;
7         for (int c = 0; c < cols ; c++) // transpose by columns
8             for (int i = 0 ; i < terms ; i++)
9                 // find and move terms in column c
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     } // end of if (terms > 0)
17     return b;
18}
```

$O(\text{terms} \cdot \text{cols})$

=====

행렬의 전치

◆ 메모리를 조금 더 사용한 개선 알고리즘 : *FastTranspose*

- 먼저 행렬 ***this**의 각 열에 대한 원소 수를 구함
 - ◆ 전치 행렬 b의 각 행의 원소 수를 결정
- 이 정보에서 전치행렬 b의 각행의 시작위치 구함
- 원래 행렬 a에 있는 원소를 하나씩 전치 행렬 b의 올바른 위치로 옮김

	ROW_SIZE	ROW_START
[0]	2	0
[1]	1	2
[2]	2	3
[3]	2	5
[4]	0	7
[5]	1	7
	↑	↑

of terms starting position
in b's row (a's col) of b's row

- 실행시간: $O(\text{cols} + \text{terms})$

STRING 추상 데이터 타입

- ◆ 문자열(string): $S = s_0, \dots, s_{n-1}$ 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
- ◆ ADT 2.5

스트링 패턴 매치: 간단한 알고리즘

◆ 함수 Find

- 두 개의 스트링 s와 pat
- pat이 s에서 탐색할 패턴
- 호출형식: s.Find(pat)
- pat과 i번째 위치에서 시작하는 s의 부분문자열 부합될 때 인덱스 i를 반환
- pat이 공백이거나 s의 부분문자열이 아닌 경우 -1을 반환
- LengthP: 패턴 pat의 길이
- LengthS: 스트링 s의 길이
- s에서 위치 LengthS-LengthP의 오른쪽은 pat과 매치될 문자가 충분하지 않으므로 고려하지 않아도 됨

스트링 패턴 매치: 간단한 알고리즘

Program 2.15: Exhaustive pattern matching

=====

```
int String::Find(String pat)
{
    // Return -1 if pat does not occur in *this;
    // otherwise return the first position in *this, where pat begins.
    for (int start = 0; start <= Length() - pat.Length(); start++)
    {
        // check for match beginning at str[start]
        int j;
        for (j = 0; j < pat.Length() && str[start+j] == pat.str[j]; j++)
            if (j == pat.Length()) return start; // match found
        // no match at position start
    }
    return -1; // pat is empty or does not occur in s
}
```

$O(\text{LengthP} \cdot \text{LengthS})$

=====

$j == \text{pat.Length()} - 1$

스tring 패턴 매치: KMP 알고리즘

◆ string s 에서 패턴 pat 찾기

$s = s_0 s_1 \dots s_{m-1}$

$pat = a b c a b c a c a b$

$s = - a b ? ? ? \dots ?$

$pat = a b c a b c a c a b$

$s = - a b c a ? ? \dots ?$

$pat = a b c a b c a c a b$

스트링 패턴 매치: KMP 알고리즘

◆ 실패 함수 f (failure function) 정의

$$= f(j) \begin{cases} \text{제일 큰 } k (< j), \text{ 여기서 } p_0 \dots p_k = p_{j-k} \dots p_j \text{인, } k \geq 0 \text{가} \\ \text{존재하는 경우} \\ -1, & \text{그 외의 경우} \end{cases}$$

◆ 예제 패턴 $\text{pat} = \text{abcabcacab}$ 의 경우

j	0	1	2	3	4	5	6	7	8	9
pat	a	b	c	a	b	c	a	c	a	b
f	-1	-1	-1	0	1	2	3	-1	0	1

실패 함수를 사용하는 패턴 매치

```
int String::FastFind(String pat)
{ // pat가 s의 서브스트링인가를 결정
    int posP = 0, posS = 0;
    int LengthP = pat.Length(), LengthS = Length();

    while ((posP < lengthP) && (posS < lengthS))
        if (pat.str[PosP] == str[PosS]) { // 문자 매치
            posP++; posS++;
        } else
            if (posP == 0)
                posS++;
            else posP = pat.f[posP-1] + 1;
    if (posP < lengthP) return -1;
    else return posS - lengthP;
}
```

— FastFind의 복잡도: $O(\text{lengthS})$

실패 함수

Program 2.17: Computing the failure function

```
=====
1 void String::FailureFunction()
2 { // Compute the failure function for the pattern *this.
3   int lengthP = Length() ;
4   f[0] = -1;
5   for (int j = 1 ; j < lengthP ; j++) // compute f[j]
6   {
7     int i = f[j-1] ;
8     while ((*str + j) != *(str+i+1)) && (i >= 0)) i = f[i];
9     if ((*str + j) == *(str + i + 1))
10        f[j] = i + 1 ;
11     else f[j] = -1;
12  }
13 }
=====
```

— FailureFunction의 복잡도: $O(\text{lengthP})$

STRING 패턴 매치: KMP 알고리즘

◆ FailureFunction을 이용한 패턴 매치

$O(\text{LengthP} + \text{LengthS})$