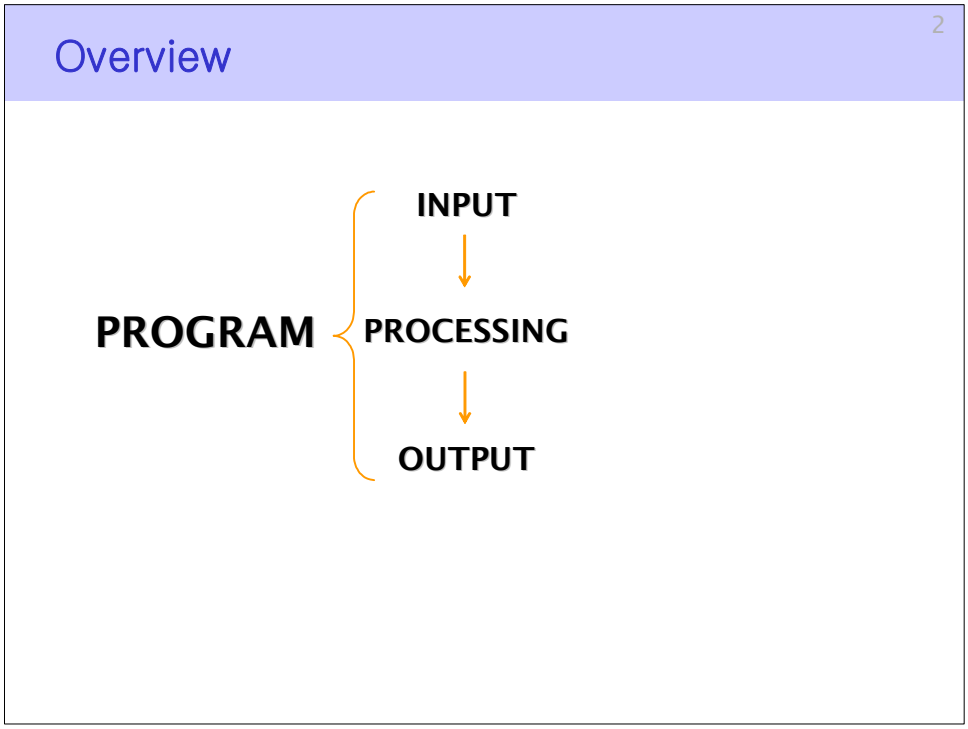
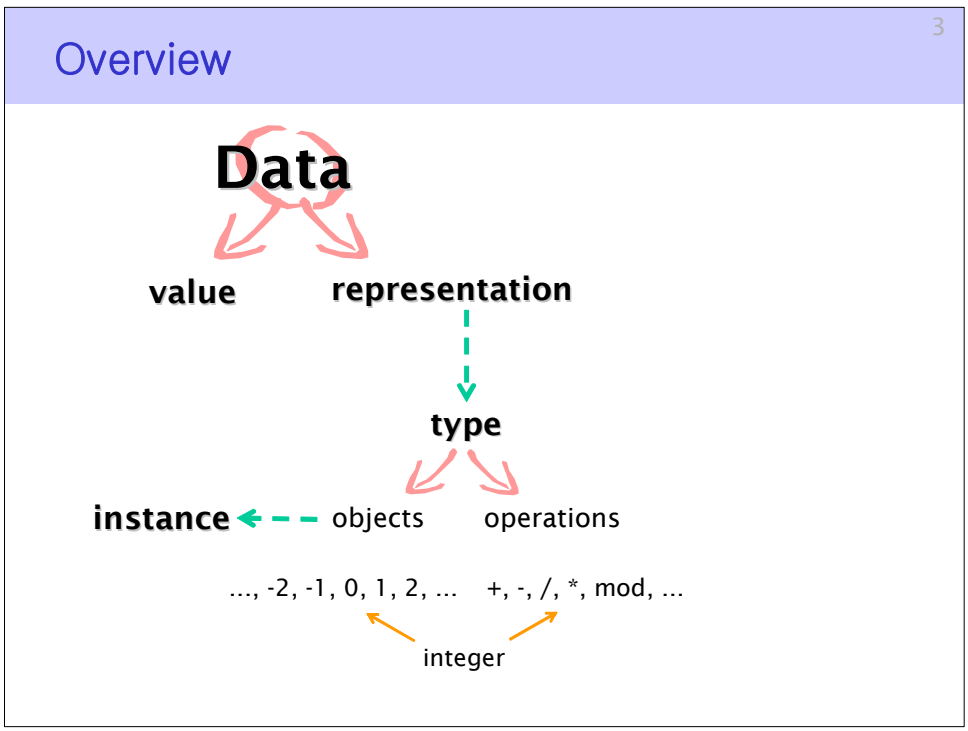
제 1 장

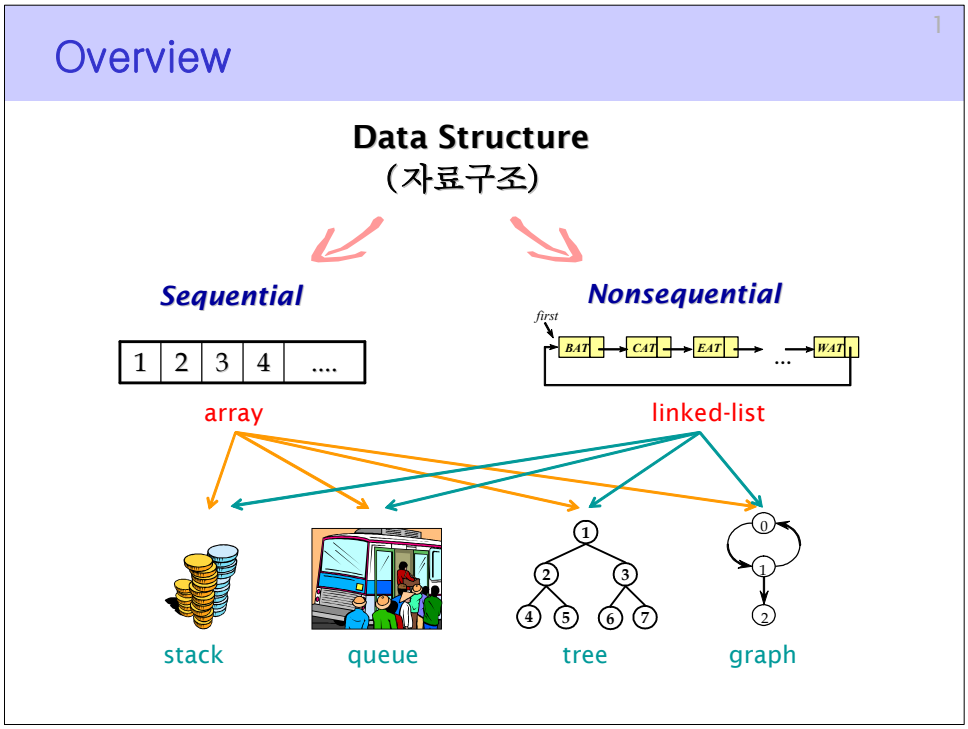
기 본 개 념





○ 데이터 타입의 명세(specification) - 데이터의 논리적 정의를 의미하며, 데이터가 무엇이고 각 연산은 무슨 기능을 수행하는가를 정의함. 데이터 타입의 의미를 정의함.

○ 데이터 타입의 구현(implementation) - 실제 데이터가 어떻게 컴퓨터의 저장장치에 표현되고 각 연산이 어떻게 구현되는가를 정의.



● 순차구조(sequential structure)

- 자료의 논리적 순서(logical sequence)가 물리적 인접성으로 표현되는 구조

○ 스택(stack)

- 데이터의 출입구가 한곳으로 지정된 자료구조.
- 먼저 입력된 데이터가 제일 마지막에 사용되며, 제일 마지막에 입력된 데이터가 먼저 사용된다.

○ 큐(queue)

- 데이터의 출구와 입구가 구분된 자료구조
- 먼저 들어온 데이터가 먼저 사용되는 구조이다.

● 비순차구조(nonsequential structure)

- 자료의 논리적 순서와 물리적 인접성이 무관한 구조로 논리적 순서를 표현하기 위한 별도의 방법이 필요하다.

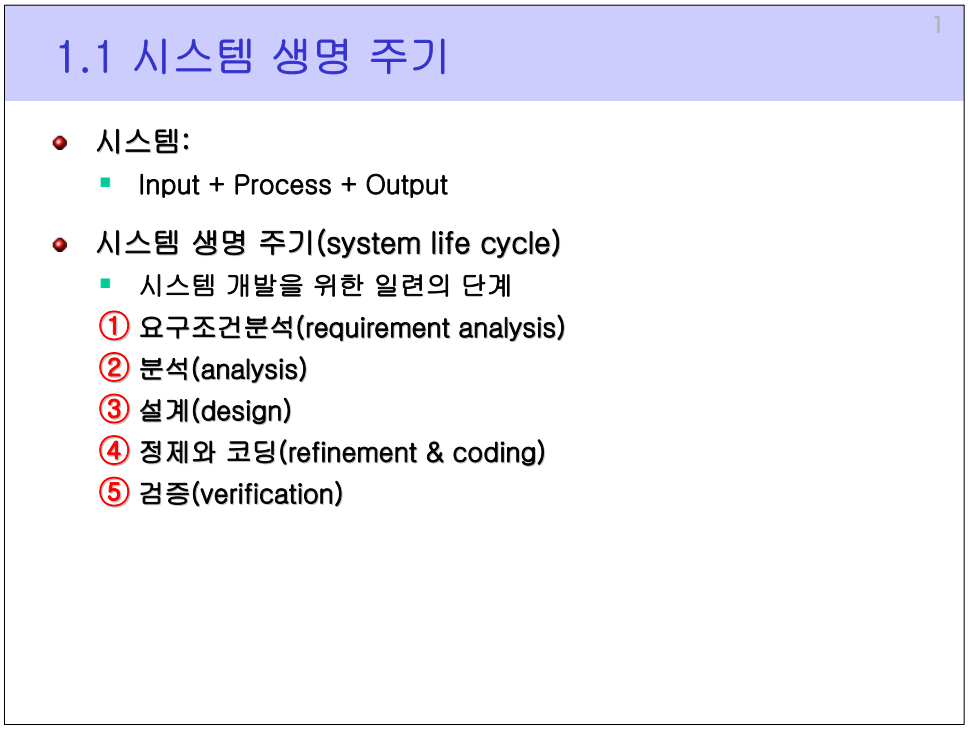
○ 트리(tree)

- 정보의 항목들이 가지(branch)로 연결될 수 있게 자료가 조직된다.
- 데이터들 간에 부모-자식(parent-child)관계를 표현하는 구조

○ 그래프(graph)

- 정점(vertex)과 이들을 연결하는 간선(edge)으로 정보를 표현하는 구조이다.

◎실생활에서 사용되는 스택, 큐, 트리, 그래프 등의 예를 생각해 보라.



○ 정확성 증명(correctness proofs)
- 보통 수학에서 사용하는 기법들을 이용해서 프로그램들의 정확성을 증명할 수 있다.
- 매우 어려우며 대형 프로젝트에서는 사용이 곤란하다.

○ 테스팅(testing)
- 테스트 데이터와 실제로 수행 가능한 코드를 이용하여 수행한다.
- 테스트 데이터는 실제로 모든 가능한 경우들이 포함되도록 신중히 구성해야 한다.

○ 오류(error) 제거
- 정확성 증명이나 시스템 테스트를 통해 발견된 오류가 발생한 코드를 제거한다.
- 오류 제거의 용이성은 시스템의 설계와 코딩 결정사항의 영향을 많이 받는다.
- 상세한 설명, 독립적인 구조의 프로그램이 필요함.
- 단위 테스트(unit test), 시스템 통합 테스트(system integration test)

1.2 객체지향 설계(Object-Oriented Design)

● 구조화 프로그래밍(structured programming)의 정의

A technique for organizing and coding computer programs in which ***a hierarchy of modules*** is used, each having ***a single entry and a single exit point***, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure. Three types of control flow are used: ***sequential, test, and iteration***. (*by Federal Standard 1037C, Glossary of Telecommunication Terms, 1996*)

● 구조화 프로그래밍(structured programming) 기법과의 유사점과 차이점은?

○ 유사점

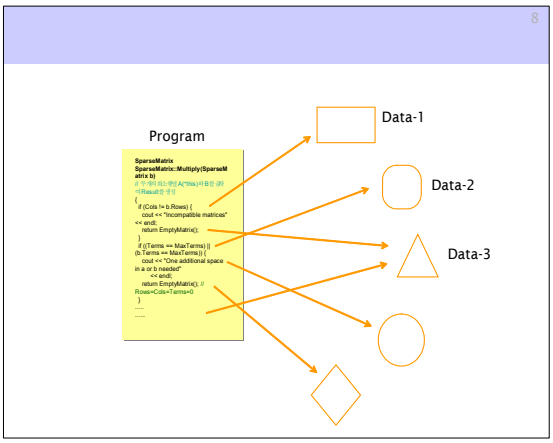
○ 차이점

7

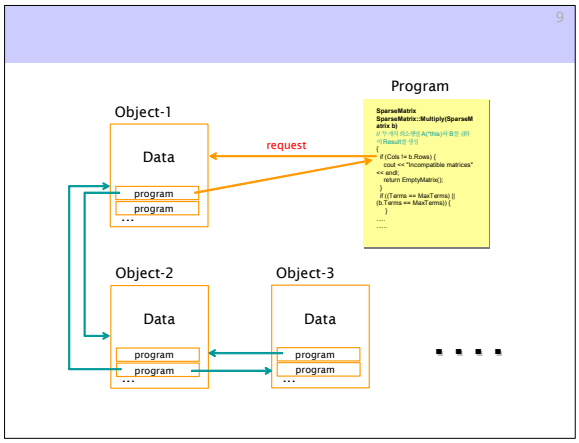
알고리즘적 분해와 객체 지향적 분해

- 알고리즘적 분해 / 기능적 분해
  - 전통적 프로그래밍 기법
  - 소프트웨어 = 프로세스
  - 프로세스의 스텝을 나타내는 모듈로 분해
    - data structure는 2차적인 문제
- 객체 지향적 분해
  - 소프트웨어 = 객체의 집합
  - 소프트웨어를 객체로 분해
    - 객체의 상호작용 = 소프트웨어
  - 재사용성(reusability)
  - 변경에 유연한 소프트웨어 시스템

8



9



1.2.2 객체 지향 프로그래밍의 기본적 정의와 개념

10

- 객체: 자신의 상태를 가지며 계산을 수행하는 개체
  - 데이타 + 연산 + Oid :
  - 데이타 : 애트리뷰트 값
- 객체 지향 프로그래밍: 구현 방법
  1. 객체를 기본적인 빌딩 블록
  2. 각 객체는 어떤 타입(클래스)의 인스턴스
  3. 클래스는 계승(inheritance)으로 연관
- 객체 지향 언어의 요건
  - 객체 지원 / 모든 객체는 클래스에 소속/ 계승 지원
- 객체 기반 언어(object-based language):
  - 객체와 클래스는 지원하되 계승은 지원 않는 언어

1.2.3 프로그래밍 언어의 발전과 C++의 역사

11

- 고급 프로그래밍 언어
  1. 제 1세대 언어 : FORTRAN수식 계산
  2. 제 2세대 언어 : Pascal, C알고리즘 표현
  3. 제 3세대 언어 : Modula, Ada
     - 추상 데이타 타입(Abstract Data Type) 지원
  4. 제4세대언어(객체지향언어) : Smalltalk, Objective C, C++
     - 계승지원
- C++ = C + OO concepts : AT&T Bell Lab.
- C의 장점
  - 효율성: 하드웨어를 효율화하는 저수준의 기능 제공
  - 유연성: 거의 모든 응용 분야에 사용
  - 가용성: 모든 컴퓨터에 컴파일러 제공

1.3 데이타 추상화와 캡슐화

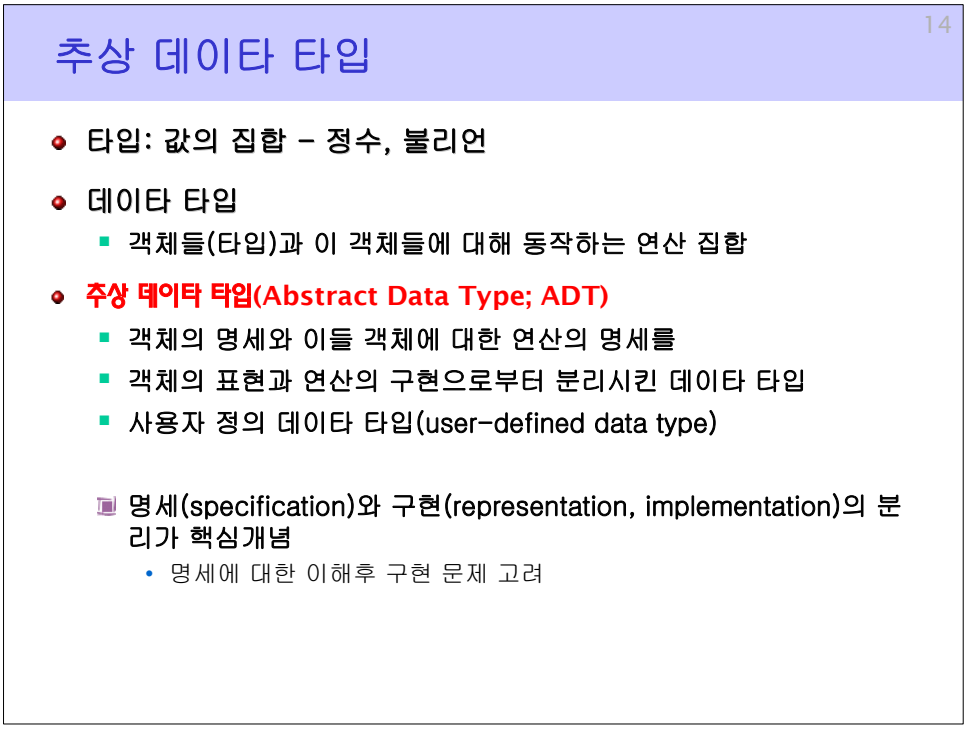
12

- 데이타 캡슐화(data encapsulation)
  - 외부에 대해 데이타 객체의 자세한 구현을 은폐
  - 내부 표현을 사용자에게 은폐 :
- 데이타 추상화
  - 데이타객체의 **명세(specification)**와 **구현(implementation)**을 분리
  - 무엇(what)과 어떻게(how)를 독립화

C++의 데이타 타입

13

- 기본데이타 타입:
  - char, int, float, double
  - 타입 수식어(modifier) : short, long, signed, unsigned
- 파생 데이타 타입:
  - 포인터(pointer) 타입, 참조(reference) 타입
- 데이타 집단화 구조:
  - 배열(array), 구조(struct), 클래스(class)
- 사용자 정의 데이타 타입



○ ADT 1.1 : 자연수 추상 데이터 타입

● 데이터 추상화와 ADT

○ 프로그램을 작성하기 위해서는 사용할 데이터의 형식과 이에 맞는 알고리즘이 필요하다. 따라서 데이터의 표현이 어떤 특성을 갖는지에 따라 주어진 알고리즘의 성능이 좌우된다고 할 수 있다.

○ 효율적이고 정확한 프로그램의 작성을 위해서는 데이터의 표현과 이에 적합한 알고리즘이 올바르게 선정되어야 한다.

○ 크고 복잡한 문제를 해결하기 위해 인간은 추상화라는 방법을 사용하여 문제를 단순화시킨 다음 해결 방법을 찾는다.

○ 데이타 추상화(data abstraction)

– 프로그램 내에 필요한 복잡한 관계의 데이터에 대해 추상화 개념을 적용하여 문제를 단순화 시킨 다음 문제 해결을 추구하는 과정.

– ADT를 이용하여 복잡한 목표 데이터를 기술하는 것. ADT에는 구체적인 표현이나 구현이 제외되어 있으므로 ADT를 기초로 알고리즘을 개발하고 데이터를 기술하면 그 과정이 매우 단순해지고 통제하기가 용이해지는 장점이 있다.

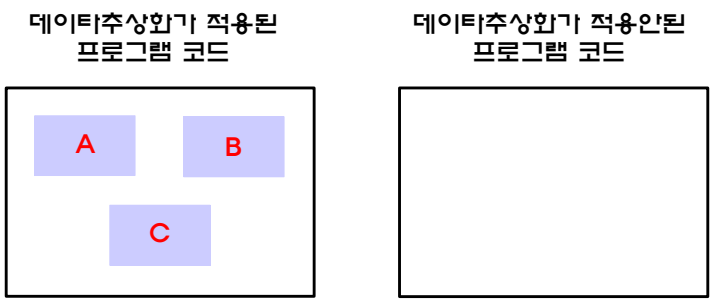
– 이 알고리즘과 프로그램, ADT와 데이터 타입과의 관계는 다음과 같이 추상화와 구체화의 관계로 표현할 수 있다.

	데이타	연산
추상화	ADT	알고리즘
구체화	데이타 타입	프로그램

데이타 추상화와 데이타 캡슐화의 장점

- 소프트웨어 개발의 간소화
  - 복잡한 작업을 부분 작업들로 분해
- 검사와 디버깅의 단순화
  - 부분 작업
- 재사용성(resuability)
  - 자료 구조가 소프트웨어 시스템에서 별개의 개체로 구현
  - 재사용
- 데이타 타입 표현의 수정
  - 데이타 타입의 내부 구현을 직접 접근하는 부분만 변경 필요
    - information hiding !
  - 그 이외 부분에는 무관
    - data encapsulation !

Figure 1.1



1.4 C++ 의 기초

19

- 프로그램의 구성
  - 헤더 화일(.h) + 소스 화일(.C)
- C++에서의 영역(scope)
  1. 화일 영역 : 함수 정의에 속하지 않은 선언
     - 클래스 정의, namespace는 화일 영역에 속함
  2. 네임스페이스 영역 :
     - 논리적으로 연관된 변수나 함수를 한 그룹으로 만드는 기법
     - 네임스페이스 std → std::cout, using std (반복적 사용시)
  3. 지역 영역 : 블록 안에서 선언된 이름
  4. 클래스 영역 : 클래스 정의에 관련된 선언
  - 변수는해당 영역내에서만 가시적임 ("영역+변수이름" 으로식별)

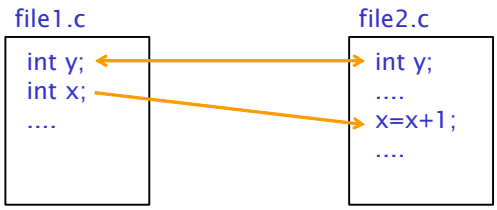
※ 영역(Scope)

19

1) 블록안에서 전역변수의이름을 지역변수로 다시 사용했을 때, 전역변수로 접근하려면 ?

2) 어떤 전역변수가 file1.C에서 정의되고 file2.C에서 사용되었다. 이 전역변수는 file2.C에서 어떻게 선언해야 하는가 ?

3) 한 프로그램에서 file1.C와 file2.C가 같은 이름의 전역변수를 정의하였다. 그러나 이 두 변수는 서로 다른 개체이다. 변수명을 바꾸지 않고 프로그램을 컴파일할 수 있는 방법은 ?



```
int w, x = 0;        // 전역변수 w, x의 정의
static int i, j;     // 정적 전역변수 i, j의 정의 (3)
f( ) {
  extern int w;      // 외부변수 w의 정의 (2)
  extern int i, j;     // 정적 외부변수 i, j의 정의
                     // extern 생략가능 - 다른 화일의 함수에서는 이용 안됨
  int x;             // 지역변수 x의 정의
  x = 1;
  {
     int x;          // 두번째 지역변수 x의 정의
     x = 2;
     w = 3;
     int y = ::x;    // 영역 연산자 ::x = 전역변수 (1)
   }
  x = 3;
}
int* p = &x;         // 전역변수 x의 주소값 치환
```

- 전역 변수(global variable)

```
int i, j
f ( ) {
  extern int i ; // 별도의 기억장소 할당 없음
  int x = i + j ;  // j : 동일 화일내의 전역변수 사용은
                   // 선언 생략 가능

}
```

- 정적 변수(static variable)

```
// 전역 변수의 영역을 동일 화일 내부로 제한

static int i, j    // 다른 화일의 함수에서는 사용 못함
f ( ) {
  extern int i ;
  int x = i + j ;
}
```

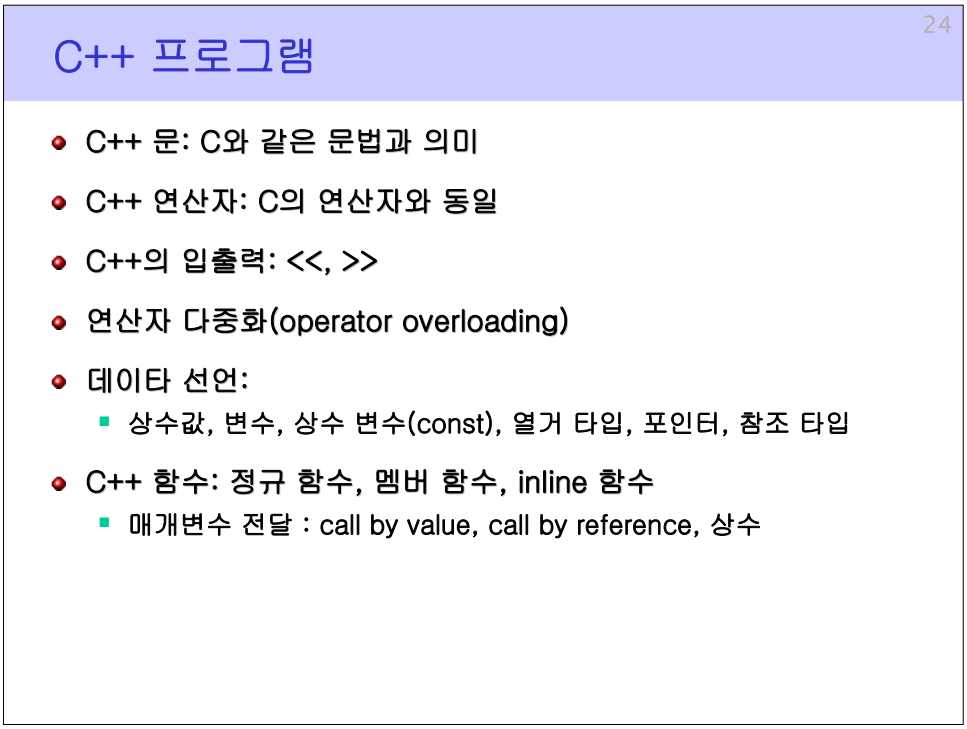
- 지역 변수의 정적 변수화

```
trystat ( )
{
  int  fade = 1 ;
  static int  stay = 1 ; // 프로그램이 적재될 때만 1로 초기화
  fade++ ;
  stay++ ;
  cout << fade << stay ;
}

...
trystat ( ) ;    2  2
trystat ( ) ;    2  3
trystat ( ) ;    2  4
...
```

fade : 호출될 때마다 1로 초기화

stay : 여러 번 호출되어 초기화 문장을 만나도 초기화는 안됨



o C++ 함수 다중화(function overloading)

- 함수의 인자 리스트(시그니쳐)가 다르기만 하면 같은 이름을 가진 함수가 둘 이상 존재 가능
- 예 :

```
int max(int, int);
int max(int, int, int);
int max(int*, int);
int max(float, int);
int max(int, float);
```

- C 언어에서 연산자 다중화의 예는?

o 인라인 함수

- 함수 정의 앞에 키워드 inline을 첨가시켜 선언
- 함수 호출을 줄임
- 매크로 함수 대신 인라인 함수 사용

```
inline int SQUARE(int x) { return (x*x); }
    // #define SQUARE(x) ((x) * (x))와 같음

inline int sum(int a, int b)
    { return a + b; }
    // 만일 i = sum(x,12); 이면 i = x+12; 으로 대체
```

1.5 알고리즘 명세

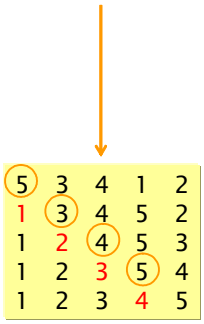
- 알고리즘(Algorithm)
  - 특정 일을 수행하기 위한 명령어의 유한집합
- 알고리즘의 요건(criteria)
  1. 입력 : (외부) 원인 >= 0
  2. 출력 : 결과 >= 1
  3. 명백성(definiteness) : 모호하지 않은 명확한 명령
  4. 유한성(finiteness) : 반드시 종료
  5. 유효성(effectiveness) : 기본적, 실행가능
- program ≡ algorithm
- flowchart ≠ algorithm
  - 명백성과 유효성의 결여

예제 : 선택정렬

- 선택 정렬
  - n >= 1 개의 서로 다른 정수의 집합을 정렬하는 프로그램을 작성

"정렬되지 않은 정수들 중에서 가장 작은값을 찾아서 정렬된 리스트 다음자리에 놓는다"

  - 초기값과 결과의 저장장소는 ?
    - *i* th integer → a[*i*-1] (1 ≤ *i* ≤ n)



예제 : 선택정렬

- 프로그램 1.5 : 선택 정렬 알고리즘

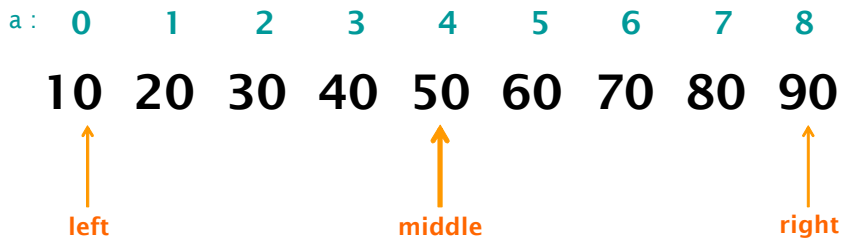
```
for (int i = 0; i < n; i++) {
    a[i]에서부터 a[n-1]까지의 정수값을 검사한 결과
        a[j]가 가장 작은값이라 하자;   // ①
    a[i]와 a[j]를 서로교환;  // ②
}
```

- 문제점
  1. 최소정수 ? : a[i]와 a[i+1] ~ a[n-1]과 비교
  2. 최소정수 a[j]와 a[i]의 교환 ? : t=a[i]; a[i]=a[j]; a[j]=t;

○ Program 1.6 : 선택정렬

이진탐색

29

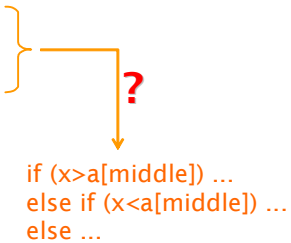


middle = ⌊(left+right)/2 ⌋

이진탐색

34

- 이진 탐색 : x = a[j]인 j를 탐색
  - left, right: 배열(a[0], a[1], ... , a[n-1])의 왼쪽, 오른쪽 끝 지점
  - 초기 값으로 left = 0, right = n-1
  - list의 중간 위치 middle = (left + right) / 2로 설정
  - a[middle]과 x를 비교할 경우 3가지 경우 중에 하나를 고려
    - x > a[middle]: left← middle+1
    - x < a[middle]: right ← middle-1
    - x = a[middle]: middle을 반환



○ Program 1.7 : 두 원소의 비교
○ Program 1.8 : 이진탐색 알고리즘
○ Program 1.9 : 이진탐색 C++함수

1.5.2 순환 알고리즘

- 순환(recursive) 함수
  - 자기 자신을 다시 호출하는 프로시져
  - (직접, 간접) : 호출 순서
  - 예: Factorial(!) : N! = N*(N−1)! = N*(N−1)*(N−2)*....*2*1*1

```
int FACT(int N)
{
   if N=0 return (1); //순환함수의 완료조건
   else return (N*FACT(N-1));
}
```

  - 실행 과정의 효율적 기술:
    - 순환적으로 정의된 자료구조
  - 실행 효율 저하
    - 실행 시간에 순환을 위한 준비 설정이 필요

예1.4 순환적 이진탐색

- 프로그램 1.10 : 이진 탐색의 순환적 구현

```
int BinarySearch(int *a, const int x, const int left,const int right)
// 정렬된 배열 a[left], ..., a[right]에서 x를 탐색
{
  if(left <= right) {
    int middle = (left + right) / 2;
    switch(compare(x, a[middle])) {
      case '>' : return BinarySearch(a, x, middle+1, right); // x > a[middle]
      case '<' : return BinarySearch(a, x, left, middle-1);  // x < a[middle]
      case '=' : return middle;                              // x == a[middle]
    } // switch의 끝
  } // if의 끝
  return -1; // 발견되지 않음
} // BinarySearch의 끝
```

○ 프로그래밍 – 선택정렬과 이진탐색 : n개의 데이터를 입력하여 정렬하고 정렬된 데이터로부터 주어진 값을 탐색하는 C++ 프로그램을 작성하라. 이때 정렬 방법은 선택정렬을 사용하고 탐색은 순환적 이진탐색 기법을 사용한다.

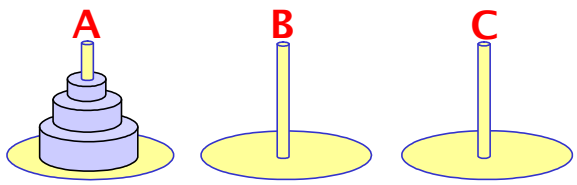
- 입력 데이터 : 터미널 또는 화일에서 입력
- 탐색 데이터 : 터미널에서 입력
  - 제일 작은 값, 제일 큰 값, 임의의 값, 없는 값의 탐색
- 결과 출력: 입력 데이터, 정렬된 데이터, 탐색된 데이터의 배열 인덱스

○ 프로그래밍 – 하노이 타워(Hanoi tower) : 3개의 막대가 있고 막대에는 서로 다른 반경을 가지는 원판이 쌓여 있다. 처음에는 각 원판이 반경이 큰 순서로 그림(a)와 같이 막대 A에 쌓여 있다. 원판을 다음 규칙에 따라 움직여서 그림 (b)와 같이 막대 C로 모두 옮기는 프로그램을 작성하라.

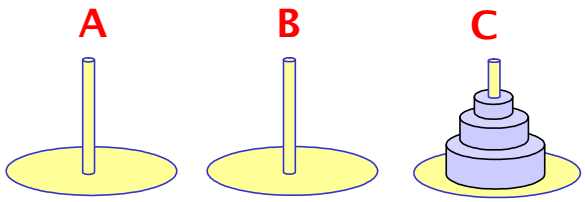
① 한 번에 한 개의 원판만을 다른 탑으로 이동할 수 있다.

② 쌓아 놓은 원판은 항상 위의 것이 아래의 것보다 작아야 한다.

(a)



(b)



- 입력 데이터 : 원판의 수를 터미널에서 입력
- 결과 출력
  - 원판의 이동 상황 및 이동 횟수 출력 (원판 이동 상황의 출력 예: 원판 4 (A →C), A: 1, 2, 3, B: 5, C: 4)
  - 원판의 수를 3개, 5개, 7개로 변경하여 프로그램을 실행하고 원판의 수에 따라 원판의 이동 횟수가 어떻게 변하는지 보이라. 또 원판의 수가 n 개 일 때 원판의 이동 횟수를 n에 관한 식으로 표현해보라.

예제

피보나치 수열(Fibonacci sequence)은 다음과 같다.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

이 수열의 각 항(term)은 바로 직전 두 항의 합으로 만들어진다. f_0를 초항이라 하고, $f_0=0$, $f_1=1$이라 할 때, 피보나치 수열의 순환 정의는 다음과 같다.

$$\begin{cases} f_0 & n=0 \\ f_1 & n=1 \\ f_n=f_{n-1}+f_{n-2} & n\geq 2 \text{ 일 때} \end{cases}$$

이 피보나치 수열의 항은 순환적으로 다시 다른 두 개의 피보나치 항으로 정의되어 있다. 이 정의에 따라, 피보나치 수열의 일반항을 구하는 순환 함수를 작성해 보면 다음과 같다.

```
fib(n)
  if n<=0 then return 0;
  if n=1 then return 1
  else return (fib(n-1) + fib(n-2));
end fib()
```

이 방법은 간단하고 이해하기 쉽지만 아주 비효율적이다. 이것을 보기 위해 순열의 6번째 숫자 5를 계산하기 위한 **fib(5)**가 어떻게 실행되는가를 그림 2.2가 보여주고 있다.

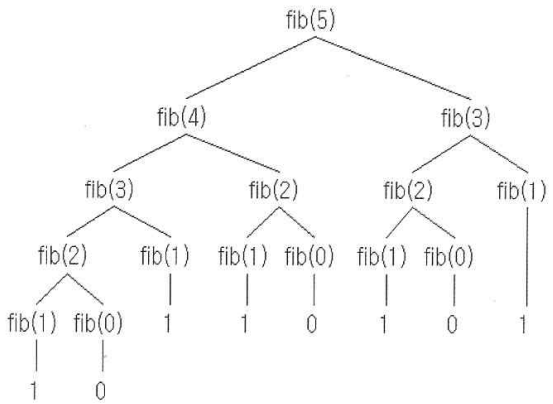
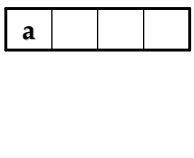


그림 2.2 fib(5)의 실행 과정

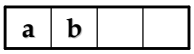
이 그림에서 알 수 있듯이 n이 5인 경우 fib(5)은 15번 호출되었다. 만일 n이 10이라면 177번, n이 25라면 242785번 fib() 함수를 호출하게 된다. 함수의 호출은 실행 시간을 상당히 지연시키기 때문에 과도한 순환 호출은 함수 실행을 비효율적으로 만드는 중요 원인이 된다. 이것은 순환적 정의가 순환적 알고리즘으로 문제를 해결하는 데 최적의 방법이라는 것을 보장하는 것이 아니라는 것을 보여주는 좋은 예이다.

예제1.5 순열

- 순열 생성기
  - perm(a, b, c, d);
    - a + perm(b, c, d)
    - b + perm(a, c, d)
    - c + perm(a, b, d)
    - d + perm(a, b, c)



  - **perm(b, c, d)**
    - b + perm(c, d)
    - c + perm(b, d)
    - d + perm(b, c)

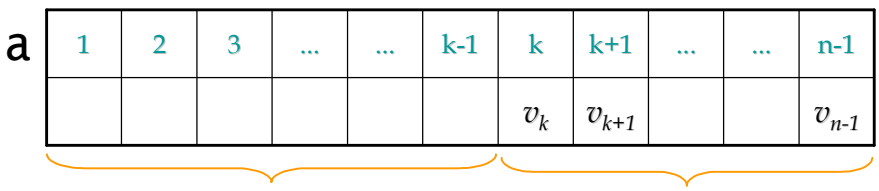


  - **perm(c, d)**
    - c + perm(d) → **a b c d**
    - d + perm(c) → **a b d c**

○ Program 1.11 : 순환적 순열생성기

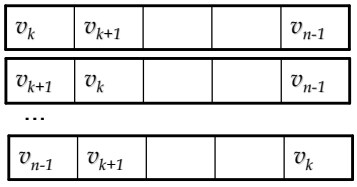
- n-1 개의 원소로 된 집합으로부터 모든 가능한 순열을 출력
- *perm(a, 0, n)* → *a[0], a[1], , a[n-1]*

void perm(char *a, const int **k**, const int n)



순열생성 완료

순열생성
a[k] ~ a[n-1]



순환 알고리즘 - PERMUTATION (1)

perm(a, 0, 4)
k = 0, n = 4

a[0]	a[1]	a[2]	a[3]
a	b	c	d

```
void perm(char *a, const int k, const int n)
// a[k], ..., a[n-1]에 대한 모든 순열을 생성
{
if (k = n-1)
 then     // 순열을 출력
   { for i = 0 to n-1 do   print(a[i]) ; }
 else     // a[k], ..., a[n-1]에 있는 여러 순열을 순환적으로 생성
   { for i = k to n-1 do { // k = 0
          // a[k] ↔ a[i]
          char temp = a[k]; a[k] = a[i]; a[i] = temp;
          // a[k+1], ..., a[n-1]에 대한 모든 순열 생성
          perm(a, k+1, n);
          // 원래 상태로 되돌리기 위해 a[k] ↔ a[i]
          temp = a[k]; a[k] = a[i]; a[i] = temp;
   }
} // end of function PERM
```

i = 0 : a[0] ↔ a[0]

a[0]	a[1]	a[2]	a[3]
a	b	c	d

perm(a, 1, 4) // perm(b, c, d)
a[0] ↔ a[0];

i = 1 : a[0] ↔ a[1]

a[0]	a[1]	a[2]	a[3]
b	a	c	d

perm(a, 1, 4) // perm(a, c, d)
a[0] ↔ a[1];

i = 2 : a[0] ↔ a[2]

a[0]	a[1]	a[2]	a[3]
c	b	a	d

perm(a, 1, 4) // perm(b, a, d)
a[0] ↔ a[2];

i = 3 : a[0] ↔ a[3]

a[0]	a[1]	a[2]	a[3]
d	b	c	a

perm(a, 1, 4) // perm(b, c, a)
a[0] ↔ a[3];

순환 알고리즘 - PERMUTATION (2)

perm(a, 1, 4)
k = 1, n = 4

a[0]	a[1]	a[2]	a[3]
a	b	c	d

```
void perm(char *a, const int k, const int n)
// a[k], ..., a[n-1]에 대한 모든 순열을 생성
{
if (k = n-1)
 then     // 순열을 출력
   { for i = 0 to n-1 do   print(a[i]) ; }
 else     // a[k], ..., a[n-1]에 있는 여러 순열을 순환적으로 생성
   { for i = k to n-1 do { // k = 1
          // a[k] ↔ a[i]
          char temp = a[k]; a[k] = a[i]; a[i] = temp;
          // a[k+1], ..., a[n-1]에 대한 모든 순열 생성
          perm(a, k+1, n);
          // 원래 상태로 되돌리기 위해 a[k] ↔ a[i]
          temp = a[k]; a[k] = a[i]; a[i] = temp;
   }
} // end of function PERM
```

i = 1 : a[1] ↔ a[1]

a[0]	a[1]	a[2]	a[3]
a	b	c	d

perm(a, 2, 4) // perm(c, d)
a[1] ↔ a[1];

i = 2 : a[1] ↔ a[2]

a[0]	a[1]	a[2]	a[3]
a	c	b	d

perm(a, 2, 4) // perm(b, d)
a[1] ↔ a[2];

i = 3 : a[1] ↔ a[3]

a[0]	a[1]	a[2]	a[3]
a	d	c	b

perm(a, 2, 4) // perm(c, b)
a[1] ↔ a[3];

순환 알고리즘 - PERMUTATION (3)

perm(a, 2, 4)
k = 2, n = 4

a[0]	a[1]	a[2]	a[3]
a	b	c	d

```
void perm(char *a, const int k, const int n)
// a[k], ..., a[n-1]에 대한 모든 순열을 생성
{
if (k = n-1)
 then     // 순열을 출력
   { for i = 0 to n-1 do   print(a[i]) ; }
 else     // a[k], ..., a[n-1]에 있는 여러 순열을 순환적으로 생성
   { for i = k to n-1 do { // k = 2
          // a[k] ↔ a[i]
          char temp = a[k]; a[k] = a[i]; a[i] = temp;
          // a[k+1], ..., a[n-1]에 대한 모든 순열 생성
          perm(a, k+1, n);
          // 원래 상태로 되돌리기 위해 a[k] ↔ a[i]
          temp = a[k]; a[k] = a[i]; a[i] = temp;
   }
} // end of function PERM
```

i = 2
a[2] ↔ a[2]

a[0]	a[1]	a[2]	a[3]
a	b	c	d

perm(a, 3, 4) // perm(d)
a[2] ↔ a[2];

i = 3
a[2] ↔ a[3]

a[0]	a[1]	a[2]	a[3]
a	b	d	c

perm(a, 3, 4) // perm(c)
a[2] ↔ a[3];

순환 알고리즘 - PERMUTATION (4)

perm(a, 2, 4)
k = 2, n = 4

a[0]	a[1]	a[2]	a[3]
a	b	c	d

```
void perm(char *a, const int k, const int n)
// a[k], ..., a[n-1]에 대한 모든 순열을 생성
{
if (k = n-1)
 then     // 순열을 출력
   { for i = 0 to n-1 do   print(a[i]) ; }
 else     // a[k], ..., a[n-1]에 있는 여러 순열을 순환적으로 생성
   { for i = k to n-1 do { // k = 2
          // a[k] ↔ a[i]
          char temp = a[k]; a[k] = a[i]; a[i] = temp;
          // a[k+1], ..., a[n-1]에 대한 모든 순열 생성
          perm(a, k+1, n);
          // 원래 상태로 되돌리기 위해 a[k] ↔ a[i]
          temp = a[k]; a[k] = a[i]; a[i] = temp;
   }
} // end of function PERM
```

i = 2
a[2] ↔ a[2]

a[0]	a[1]	a[2]	a[3]
a	b	c	d

perm(a, 3, 4) // perm(d)
a[2] ↔ a[2];

perm(a, 3, 4)
k= 3 = n-1

a[0]	a[1]	a[2]	a[3]
a	b	c	d

OUTPUT : a b c d

i = 3
a[2] ↔ a[3]

a[0]	a[1]	a[2]	a[3]
a	b	d	c

perm(a, 3, 4) // perm(c)
a[2] ↔ a[3];

perm(a, 3, 4)
k= 3 = n-1

a[0]	a[1]	a[2]	a[3]
a	b	d	c

OUTPUT : a b d c

1.6 성능 분석과 측정

- 프로그램에 대한 평가 능력을 향상
- 우수 프로그램의 기준
  - ⑴ 명세에 부합
  - ⑵ 정확히 작동
  - ⑶ 문서화
  - ⑷ 논리적 작업 단위를 위한 함수의 효과적 이용
  - ⑸ 코드의 판독용이
  - ⑹ 메모리의 효율적 사용
  - ⑺ 적절한 실행시간
- 성능 평가
  - 사전 예측 :
  - 사후 검사 :

1.6.1 성능 분석

- 성능 평가(performance evaluation)
  1. 성능 분석(performance analysis)
     - 시간과 공간의 추산
     - 복잡도 이론(complexity theory)
  2. 성능 측정(performance measurement)
     - 컴퓨터 의존적 실행 시간
- 정의

공간 복잡도(space complexity)

시간 복잡도(time complexity)

1.6.1.1 공간 복잡도

48

- **프로그램에 필요한 공간**
  - **고정공간요구 : c**
    - 프로그램 입출력의 횟수나 크기와 관계 없는 공간 요구
    - 명령어 공간, 단순 변수, 고정 크기의 구조화 변수, 상수
  - **가변공간요구 : $S_P(I)$**
    - **문제의 인스턴스, I**,에 의존하는 공간
    - 가변 공간, 스택 공간
- **공간 요구량 : $S(P) = c + S_P(I)$**
- **예제 :고정 공간 요구만을 가짐 $S_{abc}(I) = 0$**

```
float abc(float a, float b, float c)
{
    return a+b+b*c + (a+b-c) / (a+b) + 4.00;
}
```

○ 함수 abc()의 공간요구
- 함수의 인자 a, b, c와 함수의 반환 값을 저장하기 위한 공간이 필요하다.
- 함수의 인자 값에 무관하게 공간 요구량은 일정하다.

○ C++에서 배열의 전달 및 함수의 호출에 필요한 공간
- 배열의 첫번째 주소와 배열의 원소수를 전달한다.
- 함수를 호출하는 경우 함수에는 형식인자, 지역변수, 반환주소를 저장하기 위한 공간이 필요하다.

○ 프로그램 1.13 : sum의 반복함수
- 함수 sum()의 공간요구량
  - 프로그램 1.13의 함수 sum()에서는 배열의 원소수를 저장하기 위한 변수 n 과 배열의 첫 원소 a[0]의 주소를 저장하기 위한 변수 a가 필요하다. 따라서 함수 sum()의 공간요구량은 변수 n의 값과는 무관함을 알 수 있다.

○ 프로그램 1.14 : sum의 순환함수
- 함수 rsum()의 공간요구량
  - 프로그램 1.14의 함수 rsum()을 한번 호출하는 경우 형식인자 a와 n, 반환값, 반환 주소 등을 저장하기 위한 4개의 공간이 요구된다. 따라서 원소수가 n인 배열의 원소 합을 구하기 위해서는 n+1회의 함수 호출이 일어나므로 필요한 총 공간요구량은 4*(n+1)이 된다.

1.6.1.2 시간 복잡도

51

- T(P) = 프로그램 P에 의해 소요되는 시간
  = 컴파일 시간 + T_P (= 실행 시간)

$T_P(n) = C_aADD(n) + C_sSUB(n) + C_lLDA(n) + C_{st}STA(n)$

  - Ca, Cs, Cl, Cst : 각 연산을 수행하기 위해 필요한 상수 시간
  - ADD, SUB, LDA, STA(n) : 특성 n에 대한 연산 실행 횟수

프로그램 단계

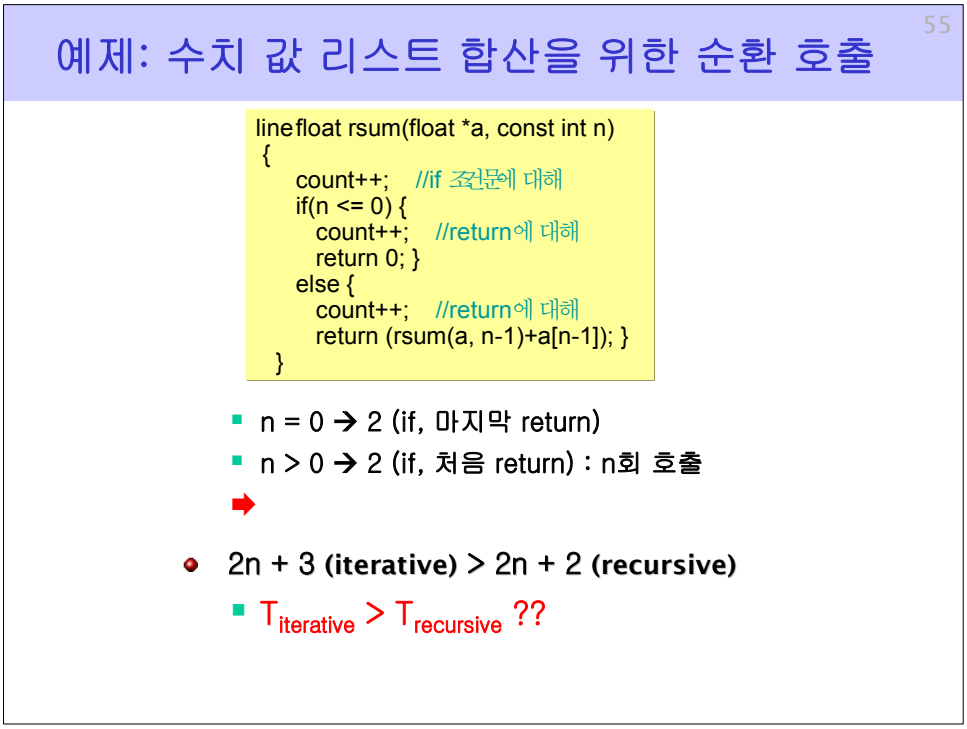
52

- 정의 : **프로그램 단계(program step)**
  - 실행 시간이 인스턴스 특성에 구문적으로 또는 의미적으로 독립성을 갖는 프로그램의 단위

    a = 2

    a = 2*b+3*c/d-e+f/g/a/b/c

- 한단계 실행에 필요한 시간이 인스턴스 특성에 독립적이어야 함
- T_P의 계산 : 시스템 클럭, 프로그램 스텝



● 단계의 계산(방법 1)

○ 전역변수 count의 사용

○ 예제 1.9: 수치 값 리스트의 합산을 위한 반복 호출
- 프로그램 1.15, 프로그램 1.16

○ 단계수의 의미
- sum이 2n+3이고 rsum이 2n+2이므로 단순히 rsum이 더 빠르다고 단정지울 수는 없다. 각 단계의 실행시간이 다를 수 있기 때문이다. 일반적으로 순환함수는 함수의 호출에 필요한 시간이 오래 걸리므로(함수의 지역변수들의 저장시간 등으로 인해) rsum이 더 많은 시간을 필요로 할 것이다.
- 그러나 동일 함수에서 n값의 변화에 따른 시간 예측은 가능하다. 즉 n이 두배가 되면 실행시간도 약 2배가 될것이라는 것을 예측할 수 있다. 실행시간은 n값에 선형적으로 비례해서 증가한다.

○ 행렬의 덧셈
- 두개의 m*n 행렬 a와 b를 합산(프로그램 1.20)
- 프로그램의 인스턴스 특성이 m과 n으로 주어진다.
- 단계수는 2mn+2m+1이 되므로 만약 m>>n이면 m과 n을 교환해서(라인 2 와 3의 for 문을 교환) 수행하는 것이 더 유리할 것이다. 이 경우 단계는 2mn+2n+1이 된다.

단계의 계산 (방법 2)

57

- 테이블 방식(tabular method) : 단계수 테이블
  1. 문장에 대한 단계수 : steps/execution, s/e
  2. 문장이 수행되는 횟수 : 빈도수(frequency)
     - 비실행 문장의 빈도수 = 0
  3. 총 단계수 = 빈도수 × s/e

예제: 수치 값 리스트 합산 위한 반복 함수

58

	문 장	s/e	빈도수	총 단계수
line	float sum(float *a, const int n)	0	0	0
1	{			
2	float s = 0;	1	1	1
3	for(int i = 0; i < n; i++)	1	n+1	n+1
4	s += a[i];	1	n	n
5	return s;	1	1	1
6	}	0	0	0
합계				***2n+3***

예제: 수치 값 리스트 합산위한 순환 함수

```
0   float rsum(float *a, const int n)
1   {
2       if (n<=0)   return 0;
3       else return (rsum(a,n-1)+a[n-1]);
4   }
```

행번호	s/e	빈 도		총 단계	
		$n=0$	$n>0$	$n=0$	$n>0$
1	0	1	1	0	0
2(a)	1	1	1	1	1
2(b)	1	1	0	1	0
3	$1+t_{rsum}(n-1)$	0	1	0	$1+t_{rsum}(n-1)$
4	0	1	1	0	0
					$2 + t_{rsum}(n-1)$

$$t_{rsum}(n) = 2 + t_{rsum}(n-1)$$
$$= 2 + (2 + t_{rsum}(n-2))$$
$$= 2 + (2 + (2 + t_{rsum}(n-3)))$$
$$\ldots\ldots$$
$$= 2n + t_{rsum}(0)$$
$$= 2n + 2$$

61

1.6.1.3 점근 표기법(O,Ω,Θ)

- Step count
  - Worst case step count : 최대 수행 단계수
  - Average step count : 평균 수행 단계수
  - ➡동일 기능의 두 프로그램의 시간 복잡도 비교
  - ➡인스턴스 특성의 변화에 따른 실행 시간 증가 예측
- 정확한 단계의 계산은 무의미
  - A의 단계수 = 3n + 3, B의 단계수 = 100n + 10 → $T_A << T_B$
  - A ≈3n (or 2n), B ≈100n (or 80n, 85n) → $T_A << T_B$

62

- 근사치 사용
  - C_1과 C_2가 음이 아닌 상수일때
  - $C_1n^2 \le T_p(n) \le C_2n^2$ 또는 $T_Q(n,m) = C_1n + C_2m$
- 예) $C_1n^2 + C_2n > C_3n$ 충분히 큰 n에 대해
  - $C_1 = 1$, $C_2 = 2$, $C_3 = 100$

    $C_1n^2 + C_2n \le C_3n$, $n \le 98$

    $C_1n^2 + C_2n > C_3n$, $n > 98$
  - 손익분기점(break even point) :

Big "Oh"

63

- 정의 [Big "oh"] $f(n) = O(g(n))$

  $f(n) = O(g(n))$ iff

  $\exists c, n_0 > 0$, *s.t.* $f(n) \le c \cdot g(n)\ \forall n, n \ge n_0$

- 예제
  - $n \ge 2$, $3n + 2 \le 4n$ → $3n + 2 = O(n)$
  - $n \ge 3$, $3n + 3 \le 4n$ → $3n + 3 = O(n)$
  - $n \ge 10$, $100n + 6 \le 101n$ → $100n + 6 = O(n)$
  - $n \ge 5$, $10n^2 + 4n + 2 \le 11n^2$ → $10n^2 + 4n + 2 = O(n^2)$
  - $n \ge 4$, $6*2^n + n^2 \le 7*2^n$ → $6*2^n + n^2 = O(2^n)$

Big "Oh"

64

- 예제
  - $n \ge 2$, $3n + 3 \le 3n^2$ → $3n + 3 = O(n^2)$
  - $n \ge 2$, $10n^2 + 4n + 2 \le 10n^4$ → $10n^2 + 4n + 2 = O(n^4)$
  - $n \ge n_0$인 모든 n과 임의의 상수 c에 대해

    $3n + 2 \le c$ 가 false인 경우가 존재하면 $3n+2 \ne O(1)$

    $10n^2 + 4n + 2 \ne O(n)$

  $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

- $f(n) = O(g(n))$
  - $n \ge n_0$ 인 모든 n에 대해 g(n) 값은 f(n)의 상한값
  - g(n)은 조건을 만족하는 가장 작은 함수여야 함

○ Big-Oh 표현

- O(1) 연산시간이 상수임을 의미하며 O(n)은 선형, $O(n^2)$은 평방형, $O(2^n)$은 지수형(exponential)이라 한다.

Omega

65

- 정의 [Omega] f(n) = Ω(g(n))
  f(n) = Ω(g(n)) iff
  $c, n_0 > 0$, *s.t.* $f(n) \geq c \cdot g(n)$ $\forall n$, $n \geq n_0$
- 예제
  - $n \geq 1$, $3n + 2 \geq 3n$ → $3n + 2 = \Omega(n)$
  - $n \geq 1$, $3n + 3 \geq 3n$ → $3n + 3 = \Omega(n)$
  - $n \geq 1$, $100n + 6 \geq 100n$ → $100n + 6 = \Omega(n)$
  - $n \geq 1$, $10n^2 + 4n + 2 \geq n^2$ → $100n^2 + 4n + 2 = \Omega(n^2)$
  - $n \geq 1$, $6*2^n + n^2 \geq 2^n$ → $6*2^n + n^2 = \Omega(2^n)$
- g(n) : f(n)의 하한 값(가능한 큰 함수)

○ Omega 표현

- $f(n) = \Omega(g(n))$에 대해 여러 함수 g(n)이 존재하지만 g(n)은 f(n)의 하한값이므로 의미를 갖기 위해서는 $f(n) = \Omega(g(n))$을 만족하는 가능한 한 큰 함수여야 한다.
- $3n+3 = \Omega(1)$, $2^n+n^3 = \Omega(n^3)$라고 표현해도 틀린 것은 아니지만 무의미한 표현이다. 따라서 $3n+3 = \Omega(n)$, $2^n+n^3 = \Omega(2^n)$이 된다.

Theta

- 정의 [Theta] $f(n) = \Theta(g(n))$
  $f(n) = \Theta(g(n))$ iff
  $C_1, C_2, n_0 > 0$, *s.t.* $C_1 \cdot g(n) \le f(n) \le C_2 \cdot g(n)$ $\forall n, n \ge n_0$
- 예제
  - $n \ge 2, 3n \le 3n + 2 \le 4n$ → $3n + 2 = \Theta(n)$: $c_1 = 3, c_2 = 4, n_0 = 2$
  - $3n + 3 = \Theta(n)$
  - $10n^2 + 4n + 2 = \Theta(n^2)$
  - $6*2^n + n^2 = \Theta(2^n)$
  - $10*\log n + 4 = \Theta(\log n)$
- g(n)이 f(n)에 대해 상한 값과 하한 값을 모두 가지는 경우
  - g(n)의 계수는 모두 1 !!

Theta

- 예제

$T_{sum} = 2n + 3$ → $T_{sum}(n) = \Theta(n)$

$T_{rsum}(n) = 2n + 2 = \Theta(n)$

$T_{add}(rows,cols) = 2rows \cdot cols + 2rows + 1$
$= \Theta(rows \cdot cols)$

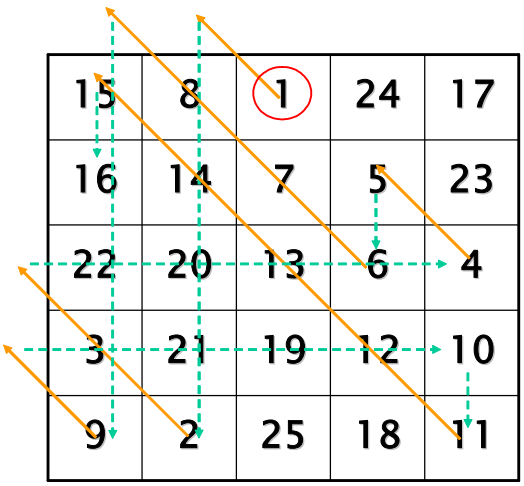
○ 표 1.4, 1.5, 1.6

매직 스퀘어

69

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

70



○ 프로그램 1.22

1.6.1.4 실용적인 복잡도

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1064	32,768	4,294,967,296

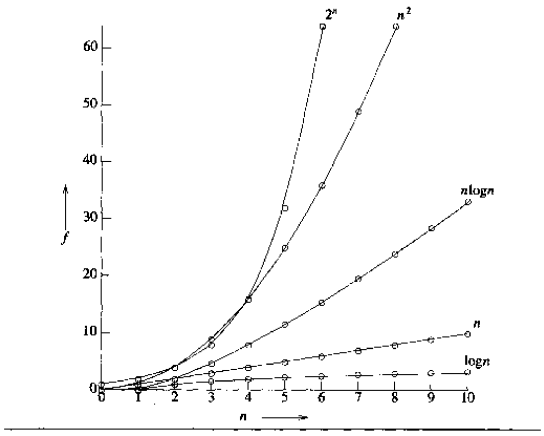


그림 1.3 : 함수 값의 그래프