Chapter 13. Recursion

박 종 혁 교수
UCS Lab
(http://www.parkjonghyuk.net)

Tel: 970-6702
Email: jhpark1@snut.ac.kr

Learning Objectives

- Recursive void Functions
  - 재귀 호출을 추적
  - 무한 재귀, 오버플로
- 값을 반환하는 재귀 함수
  - 거듭제곱 함수 pow()
- 재귀적 사고
  - 재귀적 설계기법
  - 이진 탐색

Introduction to Recursion

- 재귀 함수는 자기자신을 호출
  - *Recursive* 하다고 말함
  - 함수 정의에서 같은 함수를 호출
- C++ 은 recursion을 허용
  - 대부분의 High-Level 언어에서 사용됨
  - 유용한 프로그래밍 기술
  - 제한이 있음

Recursive void Functions

- 분할 정복 기법 (Divide and Conquer)
  - 기본 설계 기술
  - 한 task를 subtask들로 나누어서 처리
- 한 task를 작은 단위인 subtask로 나눌때 recursion!

Recursive void Function Example

- 리스트에서 원하는 값을 검색
  - Subtask 1: search 1st half of list
  - Subtask 2: search 2nd half of list
- 한 Subtask는 list의 절반씩 검색
- 재귀함수 호출로 더 작은 단위의 subtask 생성

Recursive void Function: Vertical Numbers

- 숫자를 한 줄에 하나씩 출력하기
- Example call:
  writeVertical(1234);
  Produces output:
  1
  2
  3
  4

Vertical Numbers: Recursive Definition

- 두 가지 경우를 고려하여 재귀함수 정의
- 기본적인 경우: if n<10
  - 변수 n을 화면에 출력
- 재귀적인 경우: if n>=10, 2 subtasks:

  1st subtask : 마지막 자리를 제외한 숫자를 출력

  2nd subtask : 마지막 자리의 숫자를 출력
- Example: 1234
  - 1st subtask : 1, 2, 3 출력
  - 2nd subtask : 4 출력

writeVertical Function Definition

```
void writeVertical(int n)
{
    if (n < 10)                    //Base case
        cout << n << endl;
    else
    {                              //Recursive step
        writeVertical(n/10);
        cout << (n%10) << endl;
    }
}
```

writeVertical 함수의 호출 추적

writeVertical(123);
→ writeVertical(12); (123/10)
→ writeVertical(1); (12/10)
→ cout << 1 << endl;
cout << 2 << endl;
cout << 3 << endl;

- writeVertical 함수는 2회 자기자신을 호출
- writeVertical(1)의 호출을 마지막으로 값을 출력하고 종료

Recursion Big Picture

- 성공적인 재귀함수의 개요
  - 원래 작업을 subtask로 나누어 해결하기 위해 하나 이상의 재귀 함수를 호출하는 경우
    - "recursive case(s)"
  - 함수가 재귀 함수의 호출 없이 작업을 수행하는 하나 이상의 가지 경우
    - "base case(s)" or stopping case(s)

무한 Recursion

- Base case는 반드시 입력해야함
- Base case가 없다면 무한 재귀
  - 재귀 호출이 끝나지 않음!
- Recall writeVertical example:
  - Base case는 한자리 숫자일때
  - 재귀 호출이 정지

무한 Recursion Example

- 다음 함수 정의를 고려

```
void newWriteVertical(int n)
{
        newWriteVertical(n/10);
        cout << (n%10) << endl;
}
```

- 문제가 없어 보임
- "base case“가 없음!
- 재귀가 멈추지 않는다.

Stacks for Recursion

- 스택(Stack)
  - 특별한 메모리 구조
  - 새로운 것이 상위
  - 상위부터 출력
  - "last-in/first-out”
- 재귀는 스택을 사용
  - 재귀 호출은 스택에 배치
  - 하나의 호출의 완료되면 마지막 호출 부분이 스택에서 제거

Stack Overflow

- 스캑의 크기는 제한적
  - 메모리는 유한
- 재귀 호출의 긴 체인을 지속적으로 스택에 추가
  - 계속된 추가는 base case의 삭제를 유발
- 한도 이상의 스택 사용 초과:
  - Stack overflow error
- 무한 재귀가 발생

재귀(Recursion) vs 반복(Iteration)

- 재귀가 항상 필요하진 않음
- 일부 언어에서는 재귀 허용 안됨
- 재귀를 사용하지 않고도 동일한 기능 수행 가능
  - 비재귀(Nonrecursive): 반복, 루프(loop)
- 재귀:
  - 실행이 느리고, 더 많은 저장공간을 사용
  - 장점은 적은 코딩

값을 반환하는 재귀함수

- 재귀함수는 void 함수에 국한되지 않음
- 어느 형의 값이든 반환 가능
  1. 재귀 함수 호출을 통한 값의 반환
     - 더 작은 sub-problem으로 나눔
  2. 재귀 함수 호출을 하지 않는 값의 반환
     - Base case

Return a Value
Recursion Example: 제곱근(powers)

- 미리정의된 pow(): 함수 호출

  result = pow(2.0,3.0);

  – 2의 3승 반환 power 3 (8.0)
  – 두개의 더블형 변수 입력
  – 더블형 변수 반환

Function Definition for power()

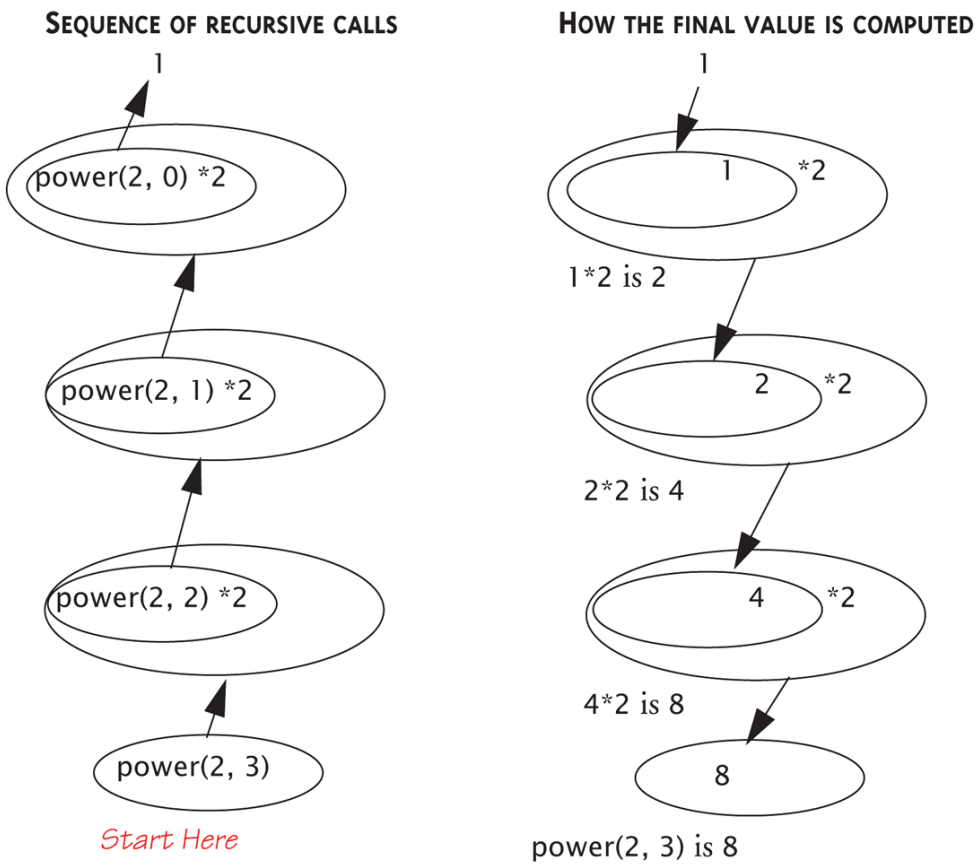
- ```
  int power(int x, int n)
  {
      if (n<0)
      {
          cout << "Illegal argument";
          exit(1);
      }
      if (n>0)
          return (power(x, n-1)*x);
      else
          return (1);
  }
  ```
  ```

Calling Function power()

- Example calls:
- power(2, 0);
  → returns 1
- power(2, 1);
  → returns (power(2, 0) * 2);
  → returns 1

Tracing Function power(): **Display 13.4** Evaluating the Recursive Function Call power(2,3)

Display 13.4 Evaluating the Recursive Function Call power(2,3)



재귀적인 개념

- 세부사항 무시
  - 스택의 동작 방법 모름
  - 추상화 원칙
  - 캡슐화 원칙
- 컴퓨터가 세부적인 작성
  - 프로그래머는 큰 그림 작성

재귀적인 개념: power

- power() 함수를 다시 한번 고려해 보자
- power ()함수의 재귀적 정의:
  power(x, n)

  returns:

  power(x, n – 1) * x
  - 올바른 수식을 보장
  - 반드시 base case 가 실행됨

재귀 디자인 기술

- 전체적인 재귀 순서를 확인할 필요가 없음
- 다음 3가지 속성 확인
  1. 무한 재귀 없음
  2. 정지하는 case 올바른 값을 반환
  3. 재귀호출시 올바른 값을 반환

재귀 디자인 체크: power()

- power()함수의 3가지 속성 확인:
  1. 무한 재귀:
     - 2nd 인수는 각 호출시 감소
     - 결국 base case에 도달
  2. 정지하는 case:
     - power(x,0) is base case
     - Returns 1, which is correct for x^0
  3. 재귀 호출:
     - For n>1, power(x,n) returns power(x,n-1)*x
     - Plug in values → correct

이진 탐색

- 배열을 검색하는 재귀 함수
- 배열은 정렬되어있다고 가정
- 배열을 2분화
  - 2분화한 첫번째 배열과 두번째 배열 중 결정
  - 그런 다음 다시 그 절반을 검색
    - 재귀적으로!

Display 13.5
Pseudocode for Binary Search

Display 13.5 Pseudocode for Binary Search

```
int a[Some_Size_Value];
```

ALGORITHM TO SEARCH a[first] THROUGH a[last]

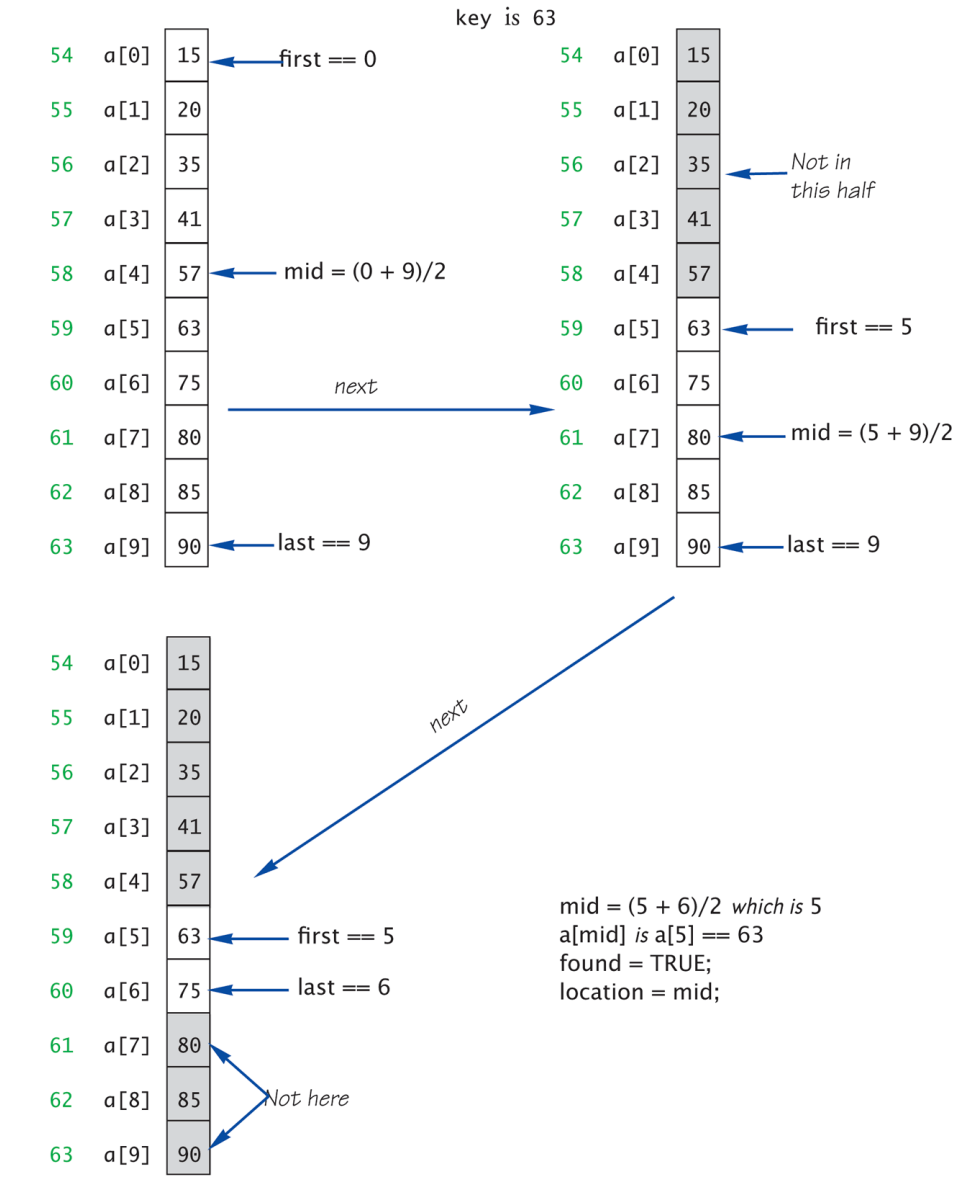
```
//Precondition:
//a[first]<= a[first + 1] <= a[first + 2] <=... <= a[last]
```

TO LOCATE THE VALUE key:

```
if (first > last) //A stopping case
    found = false;
else
{
    mid = approximate midpoint between first and last;
    if (key == a[mid]) //A stopping case
    {
        found = false;
        location = mid;
    }
    else if key < a[mid] //A case with recursion
        search a[first] through a[mid - 1];
    else if key > a[mid] //A case with recursion
        search a[mid + 1] through a[last];
 }
```

Execution of Binary Search: **Display 13.7** Execution of the Function search

Display 13.7 Execution of the Function search



이진 검색의 효율성

- 매우 빠름
  - 순차 검색 비교
- 배열의 절반은 시작에 제거!
  - 기본적으로 각각의 호출로 절반을 제거
- Example:
  Array of 100 elements:
  - 이진 검색으로 7이상의 비교 필요 없음!
    - 로그 효율 (log n)

Recursive Solutions

- 이진 검색 알고리즘은 실제로 "더 일반적인" 문제를 해결
  - 본 목표:전체 배열을 검색
  - Our function: 특정 배열 사이 검색
    - 처음과 끝의 범위가 지정됨
- 매우 일반적인 재귀 함수 디자인

요약

- 동일한 문제의 작은 인스턴스로 문제 감소 -> 재귀 solution
- 재귀 알고리즘은 두 case를 갖음:
  - Base/stopping case
  - Recursive case
- 무한재귀가 발생하지 않게 보장하지 않음
- 재귀 함수를 결정하는 기준 사용
  - 세가지 필수 속성
- 전형적으로 " 더 일반적인" 문제 해결

Q&A