PUSAN NATIONAL UNIVERSITY

REPORT

프로그래밍언어론 과제5

과 목 명 :	프로그래밍언어론
지도교수 :	
학 과 :	전기컴퓨터공학부 정보컴퓨터공학전공
학 번 :	
이 름 :	장 수현
제 출 일 :	2020년 5월 14일

부산대학교

1. Give the accessing formula for computing the location of component A [I, J] of a matrix A declared as:

A: array [LB1..UB1, LB2..UB2]

where A is stored in column-major order.

sol)

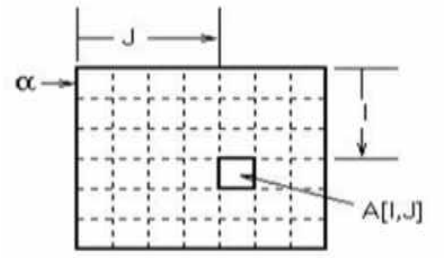

그림 1. 개념적 2차원 배열 구조

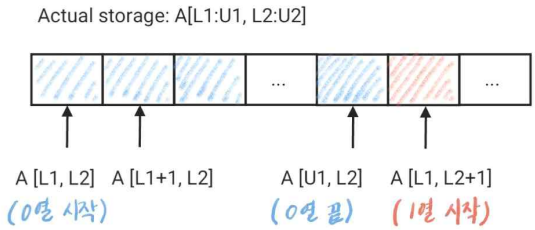

그림 2. col-major일 때, 실제 저장되는 배열의 구조

그림 1과 같이 표현되는 2차원 배열은 개념적인 구조로 메모리에 실제 저장되는 형태는 그림 2와 같이 1차원의 형태로 저장이 된다. 2차원 배열을 구현 방법으로는 row-major와 col-major가 있고, 그림 2의 경우 col-major를 나타낸 것이다. 이 메모리를 저장 형태를 따를 때, 배열의 인덱싱으로 얻게 되는 값에 대한 Accessing Formula는 다음과 같다.

L-Value(A[I, J])
= α + (J − LB_2) × S + (I − LB_1) × E
= VO + J × S + I × E

여기서,
α = base address
S = 한 열의 크기 = (UB_1 − LB_1 + 1) × E
VO = α − LB_2 × S − LB_1 × E
E = 한 element의 크기

2. For a language that allows variant records without tag fields (free-union types) such as Pascal, write a procedure

procedure GIGO (I: integer; var R: real;)

that uses a record with two variants whose only purpose is to compromise the type-checking system. GIGO takes an argument I of type integer and returns the same bit pattern as a result R of type real without actually converting the value of I to a real number.
-C 언어로 작성하되 float를 int로 변환하는 형태로 작성한다.

sol)

```c
#include <stdio.h>

union free_union {
    int INTEGER;
    float REAL;
};

float GIGO(union free_union c, int d)
{
    c.INTEGER = d;
    printf("c.INTEGER : %d\n",c.INTEGER);
    printf("c.REAL : %f\n",c.REAL);
    return c.REAL;
}

int main(void)
{
    union free_union a;
    int b = 123456789;

    int result = (int)GIGO(a,b);
    printf("result : %d\n",result);
    return 0;
}
```

C언어가 강타입 언어가 아닌 이유로 union을 말할 수 있다. union은 같은 메모리 공간에 여러 타입으로 해석 가능한 여러 타입 요소를 가짐으로써 완벽하게 타입 검사하는 것을 방해한다. 위의 코드는 union의 타입 미검사로 인해 type-checking system을 제대로 작동하지 못하게 하는 프로그램이다.

union과 int를 인자로 받는 함수 GIGO는 union 내부의 int 변수 INTEGER를 인자로 받은 int 타입의 b로 초기화한다. 그 후 INTEGER와 똑같은 비트를 가지고, 같은 메모리 공간을 공유하는 float타입의 REAL을 반환한다. INTERGER와 REAL이 공유하는 메모리 공간에 들어가 있는 특정 값을 int타입으로 해석하면 GIGO함수에 전달되었던 값인 123456789가 되지만, float타입으로 해석하면 전혀 다른 값이 된다. 그래서 의미 없는 쓰레기 값이 반환되고, 이후 int 타입으로 명시적 변환을 해주더라도 union 내부의 INTEGER가 가지는 값이 아닌 반환 받은 쓰레기 값이 int타입으로 변환되어 result에 대입된다. 이것은 union 내부 변수들에 대해 타입 검사를 하지 않음으로써 발생한 결과이다.

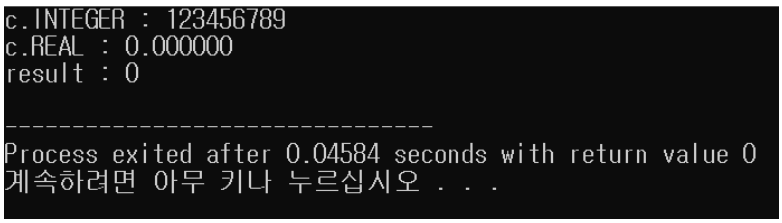

그림 3. 예시로 든 프로그램을 실행시킨 결과 화면

그림 3은 위의 프로그램을 실행시켰을 때 나오는 실행 결과화면으로 c.INTEGER와 c.REAL은 같은 메모리 공간에서 똑같은 비트 값을 가졌는데도 다른 값을 출력함을 확인할 수 있다. 그래서 결국 result에는 0이라는 의미 없는 값이 들어가게 되었다는 것도 함께 확인할 수 있다.

3. Assume that language BL includes a stack data structure and three operations:

NewTop(S, E), which adds the element E to the top of stack S,
PopTop(S), which deletes the top element of stack S,
GetTop(S), which returns a pointer to the location in stack S of the current top element.

What is wrong with the design of these three operations? How could they be redefined to correct the problem?

sol)

3.1 스택 정의

스택은 선형 데이터 구조이다. 하나의 끝에서만 요소를 삽입하거나 삭제할 수 있고, 그 끝은 Top이라 부른다. 스택은 LIFO(LAST IN FIRST OUT)의 속성을 따르며 LIFO형 자료구조라고 불리기도 한다. 스택은 다음의 Operation들을 가진다.

- Push : 스택에 요소를 삽입하는 operation.
- Pop : 항상 스택에서 가장 위에 있는 요소를 삭제하는 operation.
- Top : 항상 스택에서 가장 위에 있는 요소를 가리키는 포인터로, 초기값은 -1이다.

- Stack Full / Stack Overflow : 스택에 어떤 요소도 삽입할 수 없는 조건.
  Top의 최소값은 -1이고, 스택의 최대값은 n-1이다. (n = size of stack)
- Stack Empty / Stack Underflow : 스택에서 더 이상 요소를 삭제할 수 없는 조건.
  Top이 -1일 경우, Stack Empty.

3.2 주어진 세 operation의 문제점 및 해결 방안

주어진 세 operation은 function으로 모두 stack을 reference로 전달한다. 이는 스택의 inconsistency를 발생시킬 위험이 있다. 예를 들어 Top을 호출한 후 Pop을 하게 되면 Top의 결과로 가지고 있는 포인터는 dangling pointer가 된다. 이후 이 포인터를 참조하게 되면 치명적인 오류가 발생할 수 있다. 이를 해결하기 위해서는 stack을 class로, 각 operation을 method로 가지게 구현할 수 있다. 이로 인해 operation 호출 시에 스택을 reference로 전달하지 않아도 되고, inconsistency 문제를 해결할 수 있다. 추가적으로 GetTop method의 반환을 포인터가 아닌 Top이 가리키고 있는 값으로 수정할 수도 있다. 아래의 코드는 Java로 Stack class를 구현한 코드이다.

```
public class Stack<T> {
    private T[] data;
    private int level;
    private static final int NUM = 10;

    public Stack() {
        level = -1;
        data = (T[]) new Object[NUM];
    }

    public boolean NewTop(T data) {
        if (full()) return false;
        else {
            this.data[++(this.level)] = data;
            return true;
        }
    }

    public T PopTop() {
        if (empty()) return null;
        else {
            return data[this.level--];
        }
    }

    public T GetTop() {
        return data[this.level];
    }

    public boolean empty() {
        if (level == -1)
            return true;
        else
            return false;
    }

    public boolean full() {
        if (level == NUM - 1)
            return true;
        else
            return false;
    }

    public int size() {
        return level + 1;
    }

}
```

4. Pascal의 discriminated variant record의 문제점을 정리하고, 이 문제를 해결하기 위한 방법을 제시하라.

sol)

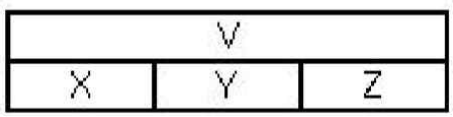

그림 4. discriminated union 객체 이미지

그림 4는 discriminated variant record인 pascal의 discriminated union 객체를 그림으로 나타낸 것이다. union 내부에는 x, y, z라는 서로 다른 타입을 가지는 세 변수가 있고, 해당 변수를 지정하는 tag V가 있다. free-union보다는 객체를 구분하는 점에서 비교적 안정적인 구조라고 할 수 있지만, 태그를 변경할 수 있기 때문에 여전히 문제가 있다. 아래 코드로 예를 들 수 있다.

```
program TestProblem;

type whichtype = (inttype, realtype, chartype);
type uniontype = record
    case V: whichtype of
        inttype: (X:integer);
        realtype: (Y:real);
        chartype: (Z:char);
    end

begin
    var P: uniontype
    P.V = inttype;
    P.X = 12345;
    P.V = chartype;
end
```

위의 코드는 태그 변수 V와 각각 int, real, char타입을 가지는 세 변수 X, Y, Z에 대한 테스트 프로그램이다. 먼저 태그를 inttype으로 지정해주었을 때 세 변수가 공유하는 메모리 공간에 있는 비트는 정수 타입으로 해석되기로 약속된다. 그래서 해당 비트를 12345라는 값으로 초기화를 해주고 변수 X를 이용해 접근할 수 있다. 그러나 태그가 chartype으로 변경되는 순간, X의 값은 더 이상 int값이 아니게 된다. char로 해석이 되기로 약속이 되고 변수 Z로만 접근이 가능하다. 그러나 해당 공간의 비트는 12345라는 정수 타입이 들어가 있으므로 Z는 무의미한 쓰레기값을 가지게 된다. 이것은 여전히 타입 검사로 잡아낼 수 없는 오류이다. 이를 해결하기 위해서는 한번 초기화된 태그는 변경하지 못하도록 방지하거나, 태그 변경 시. 해당하는 타입의 변수를 반드시 함께 초기화하도록 강제할 수 있다.