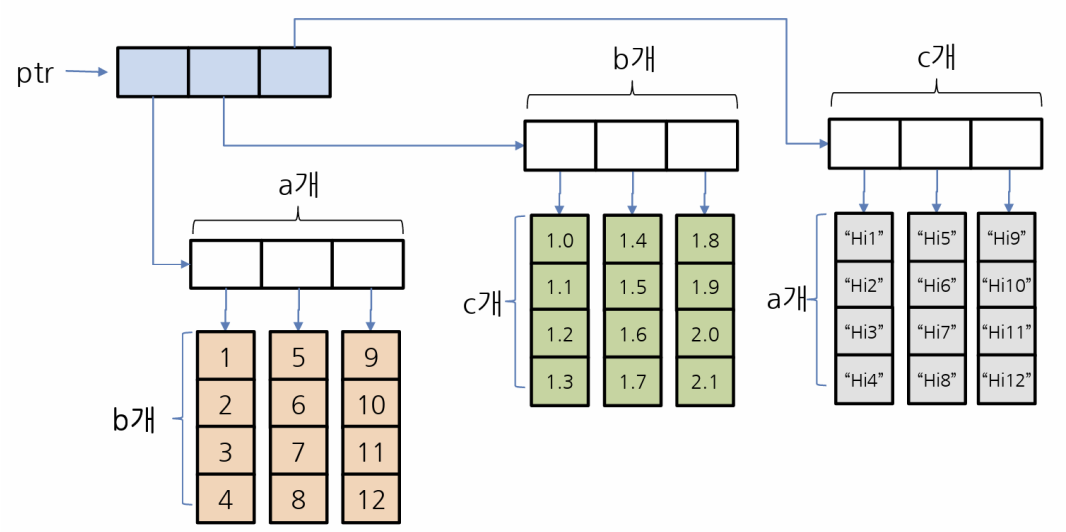
과제 1. 배열

2024-1 자료구조

과목명	자료구조	담당교수	김 승 태
학과	소프트웨어학부	학년	2학년
학번	20234748	이름	나 선 우

(1) 자료구조 만들기

[문제] 사용자에게 숫자 a, b, c를 입력 받은 후, 다음 모양을 가진 자료구조를 만들어라.



[해결 방안]

void***형의 포인터를 선언한 후, 동적으로 배열을 할당하며 역참조한다. 정수와 실수를 저장하는 부분은 동적 할당된 3차원 배열과 비슷하게 접근할 수 있으며, 문자열을 저장하는 부분은 4차원 배열로 생각할 수 있다. C에서는 문자열을 NULL 문자가 포함된 문자 배열로 처리하기 때문이다.

단, void*의 경우 직접 역참조할 수 없으므로 int*, double*, char**과 같이 적절한 타입으로 형변환하여 역참조하는 것이 필요하다. calloc으로 할당한 공간에 for문에서 순차적으로 값을 대입한 후, 메모리에 저장된 값을 ptr을 통해 순회하며 가독성 좋게 출력하면 된다.

[출력 결과]

```
a, b, c 입력: 2 3 4
<<<< ptr >>>>
ptr[0]: 000001D10DB10910, ptr[1]: 000001D10DB10930, ptr[2]: 000001D10DB0CB50

<<<< ptr[0] >>>>
ptr[0][0]: 000001D10DB10B90 = { 1, 2, 3 }
ptr[0][1]: 000001D10DB10990 = { 4, 5, 6 }

<<<< ptr[1] >>>>
ptr[1][0]: 000001D10DB0CC40 = { 1.0, 1.1, 1.2, 1.3 }
ptr[1][1]: 000001D10DB0CC70 = { 1.3, 1.4, 1.5, 1.6 }
ptr[1][2]: 000001D10DB0CCA0 = { 1.6, 1.7, 1.8, 1.9 }

<<<< ptr[2] >>>>
ptr[2][0]: 000001D10DB10890 = { "Hi1", "Hi2" }
ptr[2][1]: 000001D10DB10950 = { "Hi5", "Hi6" }
ptr[2][2]: 000001D10DB10970 = { "Hi9", "Hi10" }
ptr[2][3]: 000001D10DB10B30 = { "Hi13", "Hi14" }
```

(2) 파이썬의 List 내부 구조 파악하기

[문제] 파이썬의 List는 어떻게 구현되어 있는지 문서로 정리한다. 이 구조의 장점이 무엇인지 설명한다.

[정리 내용]

파이썬에는 모든 객체의 기본형인 PyObject 타입이 존재한다. 이것은 각 객체의 참조 카운트(ob_refcnt)와 실제 객체를 가리키는 포인터(*ob_type)로 구성된다. 길이를 가진 객체는 PyObject에서 파생된 PyVarObject를 사용하는데, 참조 카운트 및 포인터와 더불어 길이 관련 정보(ob_size)가 추가로 들어 있다. 파이썬의 리스트인 PyListObject는 PyVarObject 구조체를 포함하고 있으며, 원소들이 있는 포인터 배열에 대한 포인터(**ob_item)와 메모리에 할당된 크기(allocated)를 추가로 갖는다. 즉, 파이썬의 리스트는 다음과 같은 정보를 갖고 있다.

- ob_refcnt (참조 카운트)
- *ob_type (타입 객체를 가리킴)
- ob_size (객체의 크기)
- **ob_item (포인터 배열의 포인터)
- allocated (할당된 크기)

또한, list_resize 메서드를 통해 리스트의 길이가 할당된 메모리의 크기만큼 커지면 더 큰 메모리를 할당받는다. 리스트에 원소를 추가할 때 호출하는 append() 메서드에서도 해당 함수를 호출하여, 리스트의 크기가 할당된 크기보다 커질 경우 메모리 크기를 증가시킨다.

[장점]

일반적인 포인터가 아닌 '포인터의 포인터'를 갖고 있기에, 각 원소의 자료형이 모두 달라도 문제가 없다. 배열의 경우 모든 원소의 자료형이 같아야 하지만, 포인터의 경우 어떤 자료형의 포인터이든 그 크기는 모두 8바이트(64비트 시스템 기준)로 동일하다. 따라서 포인터 배열에 실제 데이터가 들어있는 위치를 넣고, ob_item은 포인터 배열(주소들의 배열)을 가리키면 된다. 두 번 역참조를 해야 하기 때문에 속도는 느려질 수 있지만 편의성을 크게 높일 수 있다는 장점이 있다.

또한, 리스트의 길이에 따라 메모리를 순차적으로 크게 할당함으로써 지나친 재할당으로 인한 오버헤드를 막음과 동시에 유연하게 크기를 증가할 수 있다. 이는 C++에서 사용되는 std::vector등과 유사한 사례로서, 선언 후에도 크기를 자유롭게 조절할 수 있고 자동으로 크기를 늘려준다는 장점이 있다.

(3) 다항식의 저장 방식 개선하기

[문제] 좀 더 효율적인 다항식의 저장 방식이 되려면 어떤 식으로 개선하면 좋을지 설명하고, 이를 구현하여 다항식의 덧셈과 곱셈이 진행될 수 있게 하라.

[해결 방안]

기존 코드에서는 MAX_DEGREE 만큼의 공간을 미리 할당하였으므로, 실제로 이보다 적게 공간을 사용한다면 불필요한 공간이 낭비되었다. 또한, MAX_DEGREE 보다 큰 다항식을 저장할 수 없는 문제가 생긴다. 이를 해결하기 위해 다항식의 길이를 측정하여 메모리를 동적으로 할당한다.

다항식에서 각 항의 계수와 차수를 사용자로부터 한 줄로 입력받고, strtok()을 이용해 분리한 뒤 polynominal 구조체에 저장한다. 이때 길이를 측정하여 동적으로 메모리를 할당받는다. 같은 방법으로 다른 다항식도 입력받는다.

이 프로그램은 하나의 구조체 변수가 하나의 다항식 값을 가진다. 다항식을 더하기 위해선 두 개의 인덱스를 사용하여 두 다항식을 차수의 내림차순으로 정렬하고, 같은 차수(동류항)가 있다면 더해서 저장한다. 길이를 계산하는 부분에서 두 다항식의 공통된 차수가 있는지 확인해야 하는데, 이중 반복문을 사용하는 것이 아닌 두 개의 인덱스를 사용하도록 작성하여 시간복잡도를 O(n^2)에서 O(n)으로 줄였다.

다항식의 곱셈은 우선 최대 길이 (다항식 A, 다항식 B 길이의 곱)로 메모리를 할당한 뒤 각각의 항을 곱하여 저장하고, 동류항을 찾아 더한 뒤 왼쪽으로 1만큼 쉬프트했다. 남는 공간은 realloc()을 통해 실제 사용 중인 메모리만 다시 할당받음으로써 메모리 낭비를 줄였고, sort_poly() 함수를 이용해 내림차순으로 재정렬했다.

[출력 결과]

```
Input polynominal A: 9 8 7 6 5 4 3 2
Input polynominal B: 1 5 2 4 3 3

A = 9.0x^8 + 7.0x^6 + 5.0x^4 + 3.0x^2
B = 1.0x^5 + 2.0x^4 + 3.0x^3

A + B = 9.0x^8 + 7.0x^6 + 1.0x^5 + 7.0x^4 + 3.0x^3 + 3.0x^2
A * B = 9.0x^13 + 18.0x^12 + 34.0x^11 + 14.0x^10 + 26.0x^9 + 10.0x^8 + 18.0x^7 + 6.0x^6 + 9.0x^5
```

(4) 희소 행렬 저장 방식의 곱셈

[문제] 희소 행렬 저장 방식의 곱셈을 구현하라.

[해결 방안]

1) 입력/출력 구현: 일반적인 행렬과 같이 입력을 받은 뒤, strtok()을 이용해 공백으로 구분한다. 이때 0이 아닌 칸만 행과 열을 계산하여 동적 배열로 저장한다. Ctrl+Z (EOF) 입력 시 행렬의 끝으로 판단, 입력을 중단한다. 출력 시에는 행렬의 행과 열 수를 기반으로 순회하며, 데이터가 없는 경우엔 0을 출력하고 데이터가 있는 경우엔 해당 값을 출력한다.

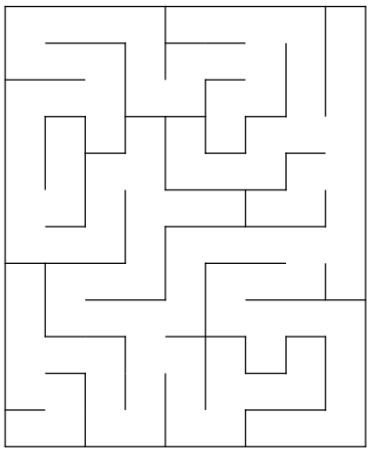
2) 저장 방식: 희소 행렬 저장 방식의 경우, 행렬 전체를 저장하는 것이 아니라 0이 아닌 부분만 '행/열/값' 형식으로 저장하여 공간을 절약할 수 있다. 이 과정에서 동적 배열을 이용하면 크기를 더욱 유동적으로 조절할 수 있다.

3) 희소 행렬의 곱셈: 우선 곱할 수 있는 행렬인지 크기를 확인하고, 각 희소 행렬을 순회하며 알맞은 위치의 수를 곱해 저장한다. 해당 위치에 값이 이미 있다면 더해서 저장하고, 없다면 새로 저장한다. 이를 반복하여 곱셈을 완료한다.

[출력 결과]

```
Input matrix (공백으로 구분, Ctrl+Z로 입력 종료)
0 0 0 0 0 0 0 1
1 1 0 0 1 0 2 0
0 0 0 0 2 0 0 1
1 1 1 0 0 0 0 0
0 0 0 2 0 0 0 0
0 0 0 0 1 0 0 0
^Z
Input matrix (공백으로 구분, Ctrl+Z로 입력 종료)
1 0 0 2
0 0 1 0
1 1 0 0
0 0 0 1
0 0 2 1
0 0 0 2
1 0 0 0
2 0 0 0
^Z
Result of (matrix A) × (matrix B)
┌ 2 0 0 0 ┐
│ 3 0 0 3 │
│ 0 0 0 0 │
│ 0 0 0 0 │
│ 0 0 0 0 │
└ 0 0 0 0 ┘
Press any key to continue . . .
```

(5) 미로 표현하기



[문제] 공간의 크기를 최소화하여 미로를 메모리에 저장하는 방법을 제안하라.

[해결 방안]

미로의 일부를 다른 코드로 치환하여 저장한다.

1. 등장하는 문자가 +, -, |, ' ' 4개이므로, 이들을 두 개씩 묶어 A, B, C, ..., P로 치환한다. 마지막 문자 '+'가 생략되나, 미로의 네 꼭짓점은 모두 '+' 문자이므로 생략하여도 충분히 추론 가능하다.

```
legend = {
    '++': 'A', '+-': 'B',
    '+|': 'C', '+ ': 'D',
    '-+': 'E', '--': 'F',
    '-|': 'G', '- ': 'H',
    '|+': 'I', '|-': 'J',
    '||': 'K', '| ': 'L',
    ' +': 'M', ' -': 'N',
    ' |': 'O', '  ': 'P'
}
```

(BFFFBFFFBCPPPOPPPOOLBFDBFDDLKPPOOPPOOOBFDLDBDLLKPPOPOPOOOLBDBBDBDDKOOOOPPPOOMEOMEMEOLLLPLPPLPKMOMMFEEMOLPLLPPLPLKMEOMFEFEOLPPLLPPPPIEFEOMFEMOLLPPLPPPLLBFDLBFBCOPPPOPPPOLBFDBBDBDKPPOPOOOOOLBDLDLBDLKPOOOOPPOOBDLDLDBFDKPOPOPPPPBFBFBFBFF)

2. 이 경우, 531바이트의 미로를 233바이트까지 줄일 수 있다. 그러나, 1~16까지의 수를 1바이트로 나타내기 때문에 치명적인 공간 낭비가 생긴다.

3. 따라서, C언어의 비트 필드 구조체를 사용하도록 한다. 4비트면 16가지를 표현할 수 있으므로, 공간을 절반 정도로 더 압축할 수 있을 것이다. 예상되는 용량은 총 *117바이트*이다.

```c
struct maze_unit {
    unsigned int p1 : 4;
    unsigned int p2 : 4;
};

FILE *fp = fopen("maze2.txt", "wb");
struct maze_unit array[117];
fwrite(array, sizeof(struct maze_unit), 117, fp);
fclose(fp);
```

비트 필드를 사용한 예시

4. 파일을 읽을 때는 이진모드로 구조체 배열을 읽어낸 다음, 이들을 "BFFF..."의 문자열로 변환한다. 그리고 위 맵을 사용하여 +---...의 미로 형식으로 변환하고, for문을 이용하여 출력한다. 19번째 문자를 출력할 때마다 '\n' 문자를 출력하면 원본 미로와 동일한 형식의 미로를 출력할 수 있다.