Pointers

프로그래밍 입문(2)

Topics

- **과제1 풀이**
- **포인터 변수 및 연산자**
- **포인터와 배열**
- 포인터와 동적할당
- 참조 변수

과제1, Q1

- 제약조건은 모양을 맞추어 ‘*’를 출력하는 것을 방지하기 위한 것.
  - cout << “   *”;
    cout << “   **”;
- 총 5줄에서 1개씩 늘어나는 별 모양을 출력하는 것.
- 공백은 반대로 1개씩 줄어듬.

```
주의사항:
cout으로 출력할 때는 오로지 STAR, SPACE, endl을
하나씩만 출력하세요.

다음의 사용은 OK.
cout << STAR;
cout << SPACE;
cout << endl;


/*
    *
   **
  ***
 ****
*****
*/
//반복문은 최대 3개까지 사용가능
```

과제1, Q1

- 우선 반복되는 5개의 줄을 위해 1개의 반복문이 필요.

- 이후 각 줄마다 공백과 별을 출력하기 위한 반복문이 2개 사용됨.

- 처음의 j반복문은 4부터 0까지 줄어들며 5~1개의 공백 출력.

- 다음의 j반복문은 0에서 4까지 늘어나며 1~5개의 별 출력.

```
/*
    *
   **
  ***
 ****
*****
*/
    //반복문은 최대 3개까지 사용가능

for(int i=0; i<5; i++) {
    for(int j=4; j>=i; j--)
        cout << SPACE;
    for(int j=0; j<=i; j++)
        cout << STAR;
    cout << endl;
}
```

과제1, Q2

- 총 3개의 줄을 출력(반복문 n)
- 공백은 2~0으로 1개씩 줄어듬 (반복문 i).
- 출력해야 하는 별의 개수는 1, 3, 5 = 홀수.
- 홀수는 보통 2n+1로 표현 (짝수는 2n).
- stars = 2*n + 1로 정의하고 이만큼 반복 (반복문j).

```
/*
    *
   ***
  *****
*/
//반복문은 최대 3개까지 사용가능
```

```
for(int n=0; n<3; n++) {
    int stars = 2*n + 1;     //각 라
    for(int i=2; i>n; i--)   //출력
        cout << SPACE;
    for(int j=0; j<stars; j++) {
        cout << STAR;
    }
    cout << endl;
}
```

과제1, Q3

- 총 5줄을 출력 (반복문 i)
- 각 줄마다 별은 1개씩만 출력:
  공백의 개수만 신경쓰면 됨.
- 3번째 줄까지는 공백이 늘어나다가, 4번째 줄부터는 줄어들어야 함.
- 5줄을 반복하는 동안 변하는 변수 i의 값에 따라 계산되는 다른 변수를 내부의 반복문j에서 조건으로 사용.

```
/*
    *
   *
  *
   *
    *
*/
    //반복문은 최대 2개까지 사용가능

for(int i=0; i<5; i++) {
    int spaces = i < 3 ? i : 4 - i;
    for(int j=0; j<spaces; j++) {
        cout << SPACE;
    }
    cout << STAR;
    cout << endl;
}
```

과제1, Q1

- 반복문 2개로 해결하는 방법.
- Q3에서는 각 줄마다 변하는 변수 i의 값에 따라 내부 반복문의 조건을 설정.
- Q1은 각 줄마다 동일하게 6개의 문자를 출력하므로 이 방법은 쓸 수 없음.
- 대신 조건에 맞추어 출력하는 문자를 변경.
  - i가 각 줄에서 출력해야 하는 공백의 숫자를 나타냄.
  - 반복시 i보다 값이 커지면 별을 출력하도록 설계.

```
/*
     *
    **
   ***
  ****
 *****
*/
  //반복문은 최대 3개까지 사용가능
```

```
char mark;
for(int i=5; i>0; i--) {
    for(int j=0; j<6; j++) {
        mark = j < i ? SPACE : STAR;
        cout << mark;
    }
    cout << endl;
}
```

포인터 변수
(Pointer Variable)

- 포인터 변수, 또는 포인터는 메모리 주소를 다루기 위한 변수.
- 포인터 선언
  - <데이터형>* <변수 이름>;
  - `int* ptr;`
- '*'은 포인터를 위한 연산자로 취급됨.

포인터 선언

- <데이터형>* <변수 이름>;
- int* ptr; OK
- int *ptr; OK
- int * ptr; OK
- int* ptr1, ptr2; → ptr1은 포인터 변수, ptr2는 int형 변수
- int *ptr1, *ptr2;

포인터 선언

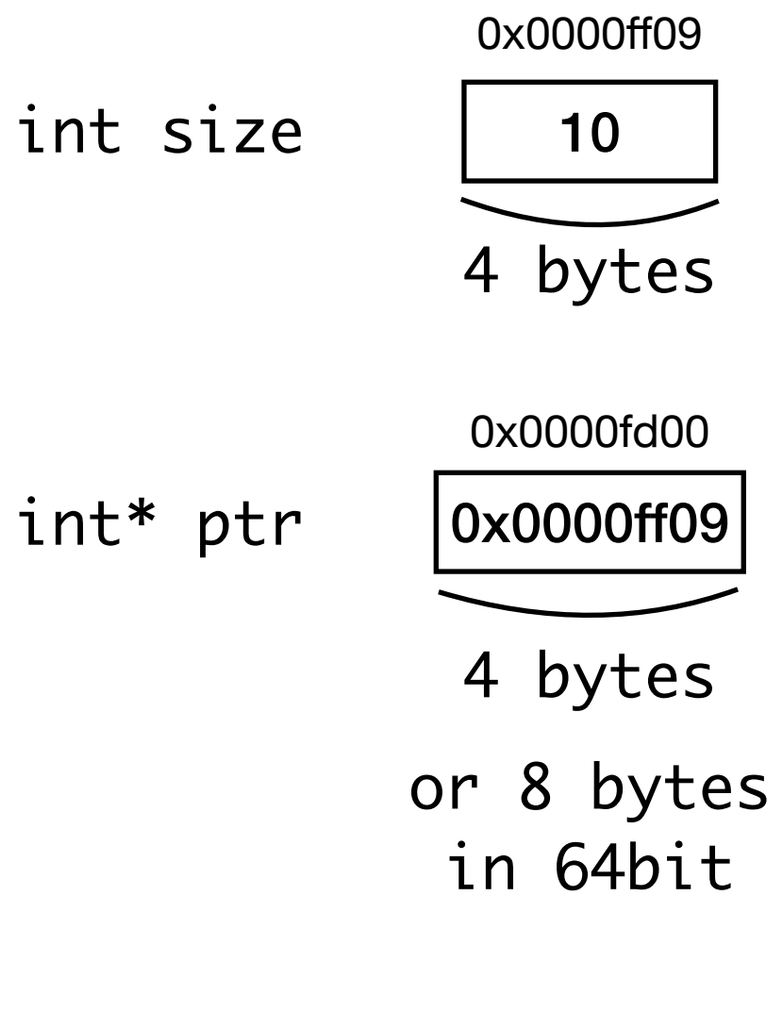
- <데이터형>* <변수 이름>;
- 다른 데이터형도 가능.
- char* str;
- double* weight;
- *연산자가 붙으면 그 데이터형의 포인터를 선언한다는 의미가 됩니다.

포인터 변수 초기화

- 선언만으로는 변수에 저장된 값의 초기화는 일어나지 않음.
- 어떤 주소를 지정하도록 초기화를 할 필요가 있음.
- 가장 기본적인 형태는 변수의 주소를 참조하는 &연산자를 사용하는 것.
- `int size = 10;//int형의 size를 선언하고 10대입`
- `int* ptr = &size;//size의 주소를 ptr에 대입`

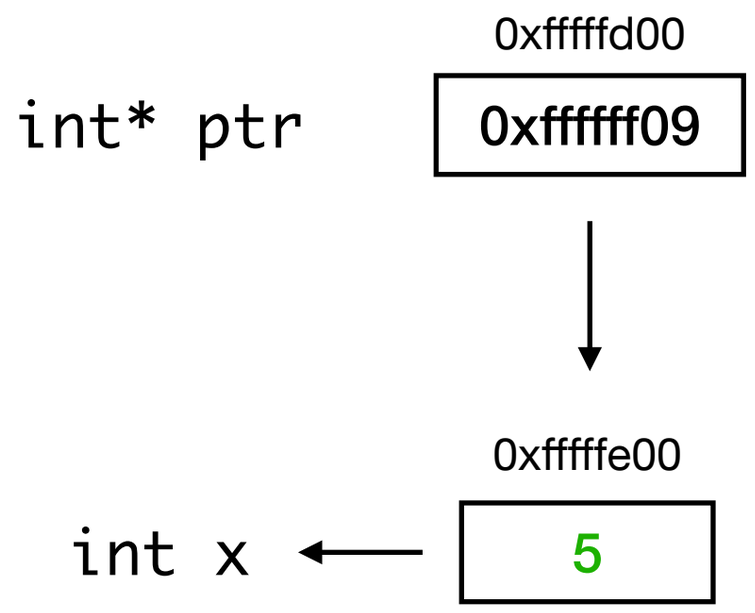
포인터 변수 초기화

- `int size = 10;`
- `int* ptr = &size;`
- &연산자를 이용해 변수의 메모리 주소값을 참조할 수 있음.
- 포인터는 그 주소값을 자신에게 할당된 메모리에 저장.



역참조
(dereference)

- `*ptr = 5;`
- `int x = *ptr;`
- x는 5의 값을 갖습니다.
- 역참조 연산자*를 이용하여 포인터가 가리키는 메모리에 있는 값을 참조.



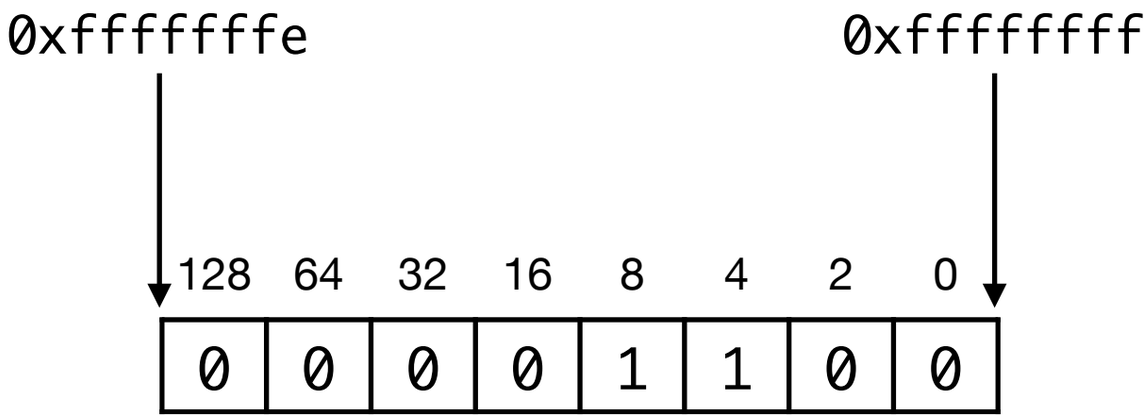
포인터 값과 메모리

- 포인터는 데이터형에 연산자 *를 붙여 정의합니다.

- 포인터는 지정한 데이터형의 메모리 주소값을 갖게 됩니다.

- 메모리는 그저 0,1만을 저장.

- 주소값에서 얼만큼 메모리를 읽을 것인지, 그 값을 어떻게 해석할 지는 데이터형에 달려있음.



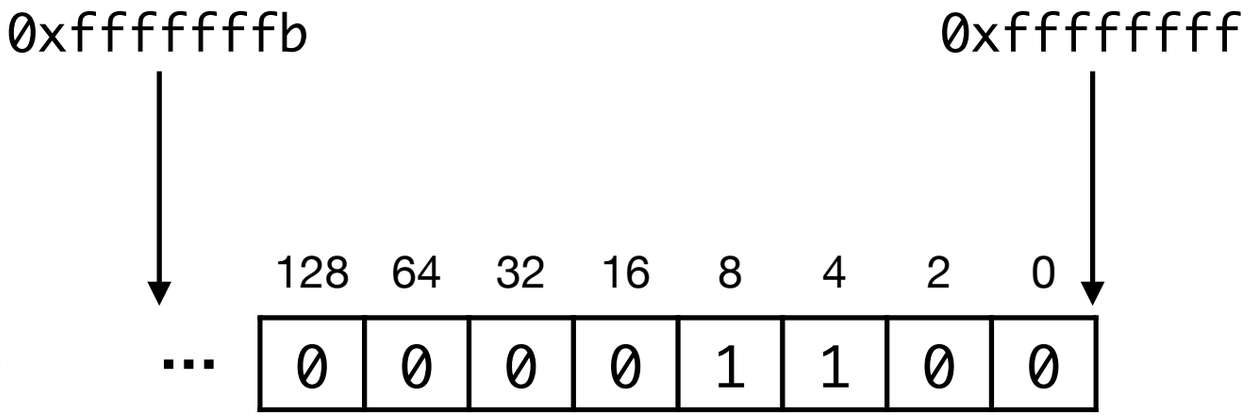

```
char c = 12;   1 byte
char* ptr = &c;
```

ptr → 0xffffffff

ptr이 char형의 포인터이므로,
0xffffffff에서 0xfffffffe까지
1 byte를 읽음.

포인터 값과 메모리

- 포인터는 지정한 데이터형의 메모리 주소값을 갖게 됩니다.
  - 보다 정확히는 시작주소값.
    - 메모리주소의 값1은 1byte 공간, int는 4 bytes이므로 값 4만큼의 공간을 점유.
- 포인터의 데이터형과 이 포인터가 참조할 메모리의 데이터형을 일치시켜 주어야 함.




```
int n = 12;    4 bytes
int* ptr = &n;

   ptr → 0xffffffff
```

ptr이 int형의 포인터이므로,
0xffffffff에서 0xfffffffb까지
4 bytes를 읽음.

포인터와 데이터형

- 일반 변수 값을 대입할 때랑 달리, short과 int, long과 int형 등은 호환이 되지 않음.
  - e.g.) `int x = 5; long y = 10;`
  - `y = x;` → OK
  - `int a = 5; int* x = &a;`
  - `long* y = &a;` → Error!
  - `y = x` → Error!

포인터와 사칙연산

- 포인터 변수에 대해서는 사칙연산 중 덧셈(+), 뺄셈(-)만 적용가능.
  - `int* ptr = &a; int* ptr2 = ptr + 1;` → OK
  - `int* ptr = &a; int* ptr2 = ptr - 1;` → OK
  - `ptr * ptr2, ptr / ptr2` → Error!
- 값 대신 정수형의 변수도 쓸 수 있음.
  - `int x = 1; long y = 1;`
  - `ptr2 = ptr + x; ptr2 = ptr - y;` → OK

포인터와 사칙연산

- 역참조의 사칙연산과는 다르다는 점을 명심하세요.
  - `int a = 4, b = 2; int* ptr = &a;`
  - `x = *ptr / b;` → `x = a / b;`
- `*ptr`은 실제로 `a`와 동일함.

포인터와 배열

- 배열 변수도 할당된 메모리의 시작 주소를 가리킴.
  - `int a[3] = { 1, 2, 3 };`
- 따라서 포인터와 호환이 가능.
  - `int* x = a;` → `x[0] = 1, x[1] = 2;`
- 배열을 가리키는 포인터의 역참조
  - `*x` → `a[0]`, `*(x+1)` → `a[1]`, `*(x+2)` → `a[2]`
  - `a[1]`은 `a`가 가리키는 주소에서 `int`형 데이터 하나만큼 이동한 주소의 값을 가져옴.
  - 이는 `int`형 포인터 `x+1`이 가리키는 주소의 값을 가져오는 것과 동일.

포인터와 배열

- 배열 변수도 *연산자를 이용한 역참조 가능.
  - *a = 1, *(a+1) = 2
- *연산자와 인덱스(index)의 복합사용.
  - *(x+1)[1] = 3 → a[2]
  - x+1은 x가 나타내는 주소에서 1만큼 이동, 여기에 다시 [1]은 1만큼 더 이동한 주소를 나타냄.

포인터와 배열

- 2차원 배열
  - `int b[2][3] = {{1, 2, 3}, {4, 5, 6}};`
    - `int** y = b;` → Error!
- `int b[2][3]`은 `int[3]`을 여럿 갖고 있는 포인터형태
  - `int`형의 포인터의 포인터인 `int**`와는 호환되지 않는다.

포인터와 매개변수

- 포인터를 매개변수로 사용해 Call by Reference로 함수를 호출.
  - `void func(int* x) { *x = 5; }`
  - `int y = 1; func(&y);`
  - &연산자 사용으로 포인터형 인자를 넘김.
  - 함수가 호출되고 나면 y = 5의 값을 갖게 됨.

포인터와 매개변수

- 1차원 배열도 마찬가지로 포인터형 매개변수로 넘길 수 있음.
    - 단 포인터형으로 넘어가면 배열의 크기 정보는 사라지므로, 변수로 크기를 같이 넘겨줘야 한다.
- 2차원 배열은?
    - `int x[3][2]; //x는 int[2]형 배열을 3개 가지고 있는 배열.`
    - 매개변수로 사용시 `int[2]`형의 포인터 형태로 넘겨 받게 된다.
    - `void func(int (*x)[2])` → `*x`로 포인터 표시(선언), `int[2]`형의 포인터.

포인터와 매개변수

- 왜 2차원 배열 매개변수를 `int x[][]` 같은 형태로 선언하면 안 될까요?
  - 포인터+1 → 포인터의 데이터형만큼 메모리주소 이동.
  - `int x[][]`일 때 x+1을 하면? → 얼마만큼 이동해야 할 지 알 수 없음.
- 실제로는 x가 2차원의 '포인터'라는 점을 명심 → 함수에서 매개변수로 받고 나면 배열처럼 그 크기를 알아낼 수 없다.

Summary

- 과제1 풀이 → 반복문 사용
- 포인터 변수의 선언과 초기화
- 역참조
- 포인터와 배열, 매개변수