Pointers

프로그래밍 입문(2)

Topics

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

포인터와 동적할당

- 배열의 동적할당에서 배웠던대로, 많은 저장공간은 실시간에 크기를 결정하는 것이 필요하거나, 효율적인 경우가 있음.

- 배열 외에도 다른 변수들을 특정 데이터형의 포인터로 선언한 후, new 키워드로 동적으로 메모리를 할당할 수 있습니다.
  - `int *ptr1 = new int;`
  - `int *ptr1 = new int(5); //값 5로 초기화`
- 동적으로 할당된 메모리의 경우, 내부적으로 처리가 달라질 뿐, 일반 포인터 변수와 동일하게 사용할 수 있음.
  - `*ptr1 = 10; //메모리에 값 대입`

new와 delete

- new로 할당하고, delete로 해제.
- C의 malloc() → new, free() → delete.
- int* ptr = new int(5);
- delete ptr;

배열의 동적할당

- new <데이터형>[<배열크기>], delete[] 사용.
- int* arr = new int[100];
- int* arr = new int[100](); //모두 0으로
- int* arr = new int[100] { 10, 20, };
  - 특정 값으로 초기화 → C++11부터 가능.
- delete[] arr;

delete로 해제?

- new로 할당되는 메모리는 운영체제(OS)에 요청하여 받은 메모리.
- 할당된 메모리는 프로그램 실행되는 동안 계속 사용하고 끝나고 나서 반환됨.
- 명시적으로 OS에 메모리를 반환하기 위해 `delete`사용.
- 이후부터는 OS가 다시 그 메모리를 다른 프로그램에 할당할 수 있음.
- **실제로 포인터의 값이나 그 안의 값이 지워지는 것이 아님을 명심할 것.**

Dangling Pointer

- delete가 실제 포인터의 주소나 그 안의 값은 바꾸지 않음.
  - `int* ptr = new int(3); //ptr=0x00…ff, *ptr=3`
  - `delete ptr; cout << ptr << endl; cout << *ptr;`
    `0x00…ff`
    `3`

- 하지만 이미 이 메모리는 OS에 반환된 상태이므로 ptr은 할당이 해제된 메모리를 가리키게 됨.

- 이런 포인터를 **Dangling Pointer**라고 부름.

Dangling Pointer

- Dangling Pointer를 다시 삭제하거나, 역참조하는 등의 동작은 정의되지 않은 동작으로 컴파일러가 제대로 된 동작을 보장할 수 없음.

- 이미 다른 프로그램에 메모리가 다시 할당되었다면 다양한 문제가 발생할 수 있습니다.

  - ```
    delete ptr; //에러 발생
    cout << *ptr; //ptr의 값을 예측할 수 없음
    *ptr = 3; //메모리 변경시 강제종료될 수 있음.
    ```

- 이를 방지하기 위해, delete시 포인터를 Null Pointer로 바꿔주는 방법을 사용할 수 있습니다.
    ```

Null Pointer

- 프로그래밍 언어에서 일반적으로 null값은 아무것도 없다는 것을 나타내기 위한 특수한 값.
- Null Pointer 또한 아무것도 가리키지 않는 포인터.
  - 0 또는 nullptr(C++11)을 포인터에 대입하여 만들 수 있음.
  - int* ptr = 0; or int* ptr = nullptr;
- 포인터를 delete하고 나서 Null Pointer로 바꿔주어 해제된 것을 표시.
- Null Pointer는 false값으로 취급되므로 포인터의 상태를 확인할 수 있습니다.
  - if(ptr)
    a = *ptr;

메모리 누수
(Memory Leak)

- 프로그램이 동작하는 중이라면,
    - 동적으로 할당된 메모리는 `delete`를 사용하여 해제해 주어야 OS에 반환됩니다.
    - `delete`를 실행하여 해제하기 전 포인터에서 주소값을 참조할 수 없게 되는 경우 메모리 누수가 발생합니다.
- 이런 메모리 누수가 반복되면 사용할 수 있는 메모리가 지속적으로 줄어들게 됩니다.
- 프로그램이 종료되기 전에는 메모리를 계속 잡고 있으므로 다른 프로그램에까지 영향을 끼치게 됩니다.

메모리 누수
(Memory Leak)

- 포인터에 메모리를 할당하고, 해제하기 전 다시 할당하는 경우.

  - ```
    int* ptr = new int; //0x00…ff
    ptr = new int; or ptr = &x;
    ```

- 0x00…ff주소의 참조는 잃어버리게 됨.

- 재할당 전 메모리 해제가 반드시 필요.

  - ```
    int* ptr = new int;
    delete ptr;
    ptr = new int;
    ```

메모리 누수
(Memory Leak)

- 메모리를 할당한 포인터 자체가 해제 전 참조할 수 없게 되는 경우.
  - if(x == y) {
    int* ptr = new int;
    ……
    }
    - 위의 코드는 포인터 ptr을 if문 안에서 선언함.
    - if문 밖에서는 이 ptr에 접근할 수 없음.
    - 만약 if문 내부에서 할당된 메모리를 해제하지 않으면, 메모리 누수가 발생.
- 똑같은 일이 반복문에서 발생한다면?

참조 변수
(Reference Variable)

- 참조 변수는 다른 변수의 별칭(alias)으로 사용하기 위해 선언하여 사용하는 변수.
- 선언 시점에 반드시 초기화가 필요.
  - <데이터형>& <변수 이름> = <참조할 변수명>
  - `int& n = a;`
  - n은 a와 동일하게 같은 변수처럼 사용할 수 있음.

참조 변수

- 일단 선언되고 나면 다른 변수를 참조하도록 변경이 불가능.
  - ```
    int a = 1;
    int b = 2;
    int& n = a;
    n = b; //n에 b의 값 2가 대입
    ```
- a에도 마찬가지로 b의 값 2가 대입.
- 값이 대입될 뿐, 참조를 바꿀 수 없습니다.
    ```

참조를 매개변수로 사용

- 참조 변수는 다른 변수의 별칭으로 사용됨 → 매개변수와 유사.
  - `func(x, y);` →
    `int func(int a,  int b) { a = …, b = … }`
-
```
int func(int& a) {
    a = 10;
}
```
- `int x = 5; func(x);` → `x = ?`
  - Call by reference이므로 x의 값도 바뀌어 `x = 10`이 됨.

참조를 매개변수로 사용

- 배열을 매개변수로 사용할 때 포인터로 변환이 일어나지 않음.
- 매개변수에 배열(avg_prices)이 들어오면 포인터로 전환되어 크기를 알아낼 수 없기 때문에, 그 크기(size)를 꼭 같이 넘겨야 함.
- 포인터형은 시작과 끝을 표시할 수 없어 Ranged For문도 사용할 수 없음.
- `void sort(double (&avg_prices)[100]);`
  - `int size = sizeof(avg_prices) / sizeof(avg_prices[0]);`
  - `for(double price : avg_prices) { … }`

참조와 포인터

- 참조는 다른 변수의 별칭 → **동일한 메모리 주소의 값**을 나타냄.
- 포인터는 **다른 변수의 메모리 주소**를 가질 수 있음.
- 포인터의 역참조 = 포인터의 메모리 주소의 값 = 참조

```
int a = 5;
int* ptr = &a;
int& n = a;
```

- 두 표현식 *ptr과 n은 동일함
  - 동일한 메모리 주소를 참조하여 그 안의 값을 나타냄.

포인터, 참조와 const

- const는 상수 - 변하지 않는 값 - 을 나타내기 위해 사용됨.
- const키워드를 포인터나 참조와 같이 사용하면 어떻게 될까요?
- 2가지 경우가 있을 수 있음.
  - 포인터, 참조의 초기화를 위해 사용되는 것이 상수.
  - 포인터, 참조 자체가 상수

포인터, 참조와 const

- 상수를 가리키는 포인터나 참조의 경우
  - `const int SIZE = 10;`
  - `const int* ptr = &SIZE;`
  - `const int& ref = SIZE;`
  - const가 없으면 에러가 발생함.
- 상수는 값을 바꿀 수 없는데, 포인터나 참조가 상수를 참조하면 상수의 값을 바꿀 수 있게 됨 → 문제를 방지하기 위해 const필요.

포인터, 참조와 const

- ```
  const int* ptr = &SIZE;
  const int& ref = SIZE;
  ```

- const가 붙은 포인터, 참조의 경우 안의 값이 상수로 취급되어 변경 불가능.
  - ```
    *ptr = 10; → Error!
    ref = 10; → Error!
    ```

- const를 사용한 포인터/참조가 변수를 가리킬 수도 있음.
  - ```
    int a = 10;
    const int* ptr = &a;
    const int& ref = a;
    ```
  - 이 경우도 *ptr, ref는 상수로 취급, 값을 변경할 수 없음.
    ```

포인터, 참조와 const

- `int* const ptr = &a;`
- 포인터 자체를 상수로 선언할 수 있습니다.
- 포인터는 메모리 주소를 갖는 변수 → 상수 포인터(const pointer)는 변하지 않는 메모리 주소를 갖게 됨.
- `ptr = &x;` → `Error!`
- `*ptr = 10` → `OK`
- `const int* const ptr = &a;` //`ptr, a` 모두 변경 불가능.

Summary

- 포인터와 동적할당
- 참조 변수
- const와 포인터, 참조