C++ 제 6강: Heap & Memory Management

SCSC 장필식

Heap

동적인 크기의 데이터를 저장할 때 사용하는 메모리 공간

Stack하고는 별도로 운영되는, 신비의 메모리 공간

Stack vs Heap

Stack

- 매우 빠름
- 할당한 메모리의 크기를 Compile-time에 미리 결정해야함

Heap

- Stack에 비해서는 느림
- 메모리 상에서 Stack과는 반대쪽에 위치함
- 프로그램의 실행 도중 원하는 만큼 메모리를 할당할 수 있음
- 메모리가 정확히 어디에 할당될지는 Memory Allocator가 자동으로 골라줌

new(할당) / delete(해체)

```
// allocate a's value in heap (and set it to 1)
int* a = new int(1);

cout << "a is: " << *a << endl; // 1
*a = 2;
cout << "a is: " << *a << endl; // 2

// deallocate a's value from heap
delete a;
```

Practice

```
int a = 2;
int* b = new int(1);
int* c = new int(a);
int* d = c;

cout << a << *b << *c << *d << endl;

delete b;
delete c;
```

좀 더 복잡한 케이스

```
void bar(int* a, int* b) {
    int* c = new int(5);
    int* d = c;
    *a = 1;
    delete c;
}
void foo(int* x) {
    int y = 10;
    int z = &y;
    bar(x, z);
}
int main() {
    int h = 3;
    int* i = new int(20);
    int* j = h;
    foo(i);
    delete i;
}
```

new 다음에는 반드시 **delete!**

```
int* a = new int(1);
// don't delete a...
```

안그리면 우리의 메모리는 더 이상 찾을 수 없는 상태가 된다.

이 메모리는 프로그램의 실행동안 계속 어딘가 공간을 잡아먹게 된다.

이런 것을 **memory leak**라고 한다.

하지만 두 번 **delete**하진 말자.

```
int* a = new int(1);
delete a;
// accidentally delete a again
delete a;
```

이미 free한 메모리를 다시 free하면 undefined behavior.

아무 일이나 일어날 수 있다. (지구 멸망을 포함해서)

Allocation in class

Person이 Heap에 저장하는 리소스를 가지고 있다고 생각해보자.

```
struct Person {
    std::string name; // 이름
    DNA* dna; // 염기서열 (매우 큼, heap에 저장할거임)
}
```

Allocation in class

근데 이 사람이 없어진다면 이 사람의 dna는 누가 delete할까?

```
struct Person {
    std::string name;
    DNA* dna;
    Person(std::string name) : name(name) {
        dna = new DNA();
    }
}

void doSomething()
{
    Person person("Adam");
    // do something with person....
}
```

delete하는걸 깜빡하지 말자

그래서 함수를 따로 만들어줬는데... 더 좋은 방법은 없을까?

다 쓰고 나서 항상 free()를 불러줘야 하는건 좀 귀찮고 까먹기 쉽다.

```
struct Person {
    std::string name;
    DNA* dna;
    Person(std::string name) : name(name) {
        dna = new DNA();
    }
    void free() {
        delete dna;
    }
}

void doSomething()
{
    Person person("Adam");
    // do something with person...
    person.free();
}
```

Destructor

Constructor하고는 반대로, 오브젝트가 제거될 때 불러지는 함수.

```
struct Person {
    std::string name;
    DNA* dna;
    Person(std::string name) : name(name) {
        dna = new DNA();
    }
    ~Person() {
        delete dna;
    }
}

void doSomething()
{
    Person person("Adam");
    // do something with person...
}

// dna data is automatically deleted after
// person is out of scope
```

RAII

어떤 클래스가 heap에 있는 리소스를 할당하고 싶으면

- constructor에서 할당(new)하고
- destructor에서 해제(delete)하도록 하자!

즉 "내 똥은 내가 치운다"

이미 우리는 **RAII** 기능을 쓰고 있었다

예전부터 계속 STL 라이브러리의 컨테이너들 (std::vector, std::list, ...)를 쓰고 있었다.

std::array를 제외한 나머지는 거의 모두 heap에 메모리를 할당한다.

근데도 우리는 신경쓸 필요가 없었다! 왜지?

```
struct StudentInfo {
    std::string name;
    double midterm, final;
    std::vector<double> homework;
};
```

이미 우리는 **RAII** 기능을 쓰고 있었다

std::vector, std::list등의 컨테이너 모두 다 알아서 리소스를 관리한다.

즉 자기 자원을 delete하는 destructor를 모두 다 가지고 있다.

클래스의 오브젝트가 없어지면 그 오브젝트들의 필드도 다 없어지기 때문에 알아서 자기의 destructor들이 불러지므로 걱정 ㄴㄴ

Synthesized Destructor

만약에 우리가 직접 destructor를 정의하지 않았을 때,

컴파일러가 알아서 비어있는 destructor를 만들어 준다.

```
struct StudentInfo {
    std::string name;
    std::vector<int> scores;

    // 컴파일러가 만드는 것들...
    // StudentInfo() : name(), scores() {}
    // ~StudentInfo() {}
}
```

(참고로 `name`과 `scores` 필드는 이 Synthesized Destructor 때문에 desctructor 것이 아니라, 오브젝트 자체가 scope에 나감에 따라 destructor가 불러지게 되는 것이다.)

Ex: Linked List (코드 예시)

Doubly Linked List

다음주까지 과제.

기존의 코드를 확장하여 앞뒤로 포인터가 존재하는 Doubly Linked List 를 만들어라.
그리고 나서 다음 함수들을 구현해라.

- 뒤에 숫자를 집어넣는 `void insertAtTail(int value)`
- 거꾸로 리스트를 프린트하는 `void traverseBackwards()`

다음 시간 예고편

근데 이거 좀 많이 귀찮다

메모리가 언제 할당되는지를 추적해서 일일이 delete해줘야 한다.

잘못하다간 어썸한 버그들을 만날 수 있다.

좀 더 편리한 방법이 없을까?

그리고 복잡하다

한 오브젝트가 new로 할당된 리소스를 들고 있으면 쉬운데,

여러 오브젝트가 하나의 리소스를 들고 있다면? 누가 delete를 해야 하는 거지?

여러 오브젝트가 여러가의 리소스를 들고 있다면? 그야말로 총체적 난국

도와줘요 **C++11** 스피드웨건

- shared_ptr / weak_ptr
- unique_ptr
- memory ownership
- 기타 등등의 메모리 관련 얘기들...

끄으읕