C++ 제 7강: Automatic Memory Management

SCSC 장필식

참고

C++ Primer 12장

중요한 내용이니 시간이 되는 사람들은 꼭 읽어보도록 하자.

Recap: Stack and 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

RAII

어떤 클래스가 heap에 있는 리소스를 할당하고 싶으면

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

즉 "내 똥은 내가 치운다"

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...

shared_ptr / weak_ptr

여러 곳에서 하나의 리소스를 "공유"하고 있을 경우.

shared_ptr

(참고: `using namespace std` 를 적용했음)

```
shared_ptr<int> p(new int(42));
shared_ptr<Person> p(new Person("Yongjae", 100));
// or
shared_ptr<int> p = make_shared<int>(42);
shared_ptr<Person> p = make_shared<Person>("Yongjae", 100);
```

shared_ptr

하나의 할당된 메모리에 여러 개의 shared_ptr가 존재할 수 있다!

```
auto p = make_shared<Person>("Yongjae", 100);
auto q(p);
```

(참고로 auto로 귀차니즘을 줄일 수 있다)

shared_ptr

근데... 메모리를 어떻게 알아서 관리하는거지?

우리의 "영재" 오브젝트는 doSomething()의 종료시점에 제거되어 있다!

```
void doSomething() {
    auto p = make_shared<Person>("Yongjae", 100);
    auto q(p);
}
```

reference counting

각 리소스마다 몇 군데에서 참조하는지를 나타내는 reference count 변수가 부여된다.
reference count가 0이 되면 메모리는 해제가 된다.

```
void doSomething() {
    auto p = make_shared<Person>("Yongjae", 100); // refcnt = 1
    auto q(p); // refcnt = 2
} // refcnt = 0 -> free!
```

reference counting

return을 하면 refcnt가 하나 증가한다.

즉 shared_ptr를 return하면 메모리가 해제되지 않는다.

```
shared_ptr<Foo> factory(T arg) {
    auto p = make_shared<Foo>(arg); // refcnt = 1
    return p; // refcnt = 2
} // refcnt = 1 (not freed yet)
void use_factory(T arg) {
    shared_ptr<Foo> p = factory(arg); // refcnt = 1
} // refcnt = 0 (now freed)
```

클래스와 **shared_ptr**

- 여기서 Image는 여러개의 Post 사이에서 공유될 수 있다.
  - 즉 메모리의 복사를 줄일 수 있다.
- 그리고 Database에서 항목이 날아가도 Post의 Image들은 유지된다.

```
struct Post {
    string title;
    string description;
    vector<shared_ptr<Image>> images;
}
struct Database {
    using int = UserId;
    map<UserId, vector<shared_ptr<Image>>> images;
}
```

Example: Text-Query Program (C++ Primer)

Example: Text-Query Program

```
element occurs 112 times
   (line 36) A set element contains only a key;
   (line 158) operator creates a new element
   (line 160) Regardless of whether the element
   (line 168) When we fetch an element from a map, we
   (line 214) If the element is not found, find returns
followed by the remaining 100 or so lines in which the word
```

```
// declaration needed for return type in the query function
class QueryResult;
class TextQuery {
public:
    using line_no = std::vector<std::string>::size_type;
    TextQuery(std::ifstream&);
    QueryResult query(const std::string&) const;
private:
    std::shared_ptr<std::vector<std::string>> file;  // input
file
    // map of each word to the set of the lines
    // in which that word appears
    std::map<std::string,
             std::shared_ptr<std::set<line_no>>> wm;
};
```

```
class QueryResult {
    friend std::ostream& print(std::ostream&, const QueryResult&);
public:
    QueryResult(std::string s,
                std::shared_ptr<std::set<line_no>> p,
                std::shared_ptr<std::vector<std::string>> f):
        sought(s), lines(p), file(f) { }
private:
    std::string sought;
    std::shared_ptr<std::set<line_no>> lines;
    std::shared_ptr<std::vector<std::string>> file;
}
```

문제

Reference counting은 좋은 방법이지만... 하나의 허점이 있다.

문제

```
struct Person {
    std::string name;
    std::shared_ptr<Home> home;
    Person(std::string name) : name(name) {}
}

struct Home {
    std::string name;
    std::shared_ptr<Person> personLiving;
    Home(std::string name) : name(name) {}
}

void doit() {
    auto person = std::make_shared<Person>("Youngjae");
    person.home = std::make_shared<Home>("SCSC");
    person.home.personLiving = person;
}
```

해결방안: **weak_ptr**

shared_ptr에 대해 참조만 하는 포인터 종류.

shared_ptr의 reference count에 영향을 주지 않는다.

```
struct Person {
    std::string name;
    std::shared_ptr<Home> home;
    Person(std::string name) : name(name) {}
}

struct Home {
    std::string name;
    std::weak_ptr<Person> personLiving;
    Home(std::string name) : name(name) {}
}

void doit() {
    auto person = std::make_shared<Person>("Youngjae");
    person.home = std::make_shared<Home>("SCSC");
    person.home.personLiving = std::weak_ptr<Person>(person);
}
```

weak_ptr

특정 shared_ptr를 참조하는 weak_ptr를 만들 수 있다.
이 weak_ptr는 lock()라는 함수를 통해서만 참조하는 shared_ptr를 접근할 수 있다.

```
auto p = make_shared<int>(42);
weak_ptr<int> wp(p);

// true if np is not null
if (shared_ptr<int> np = wp.lock()) {
    // uinside the if, np shares its object with p
}
```

unique_ptr

오직 한 군데에서 하나의 리소스를 접근하게 하고 싶은 경우

unique_ptr

```
unique_ptr<int> p(new int(1));
// or
unique_ptr<int> p = make_unique<int>(1); // in C++14
```

unique_ptr

오직 하나의 unique_ptr만 리소스를 접근할 수 있으므로,
unique_ptr를 copy하는 것은 불가능하다.

```
auto p = make_unique<int>(1);
auto q(p); // ERROR: invalid
```

unique_ptr

하지만 소유권을 옮길 수는 있다.

```
auto p = make_unique<int>(1);
unique_ptr<int> q(p.release());
```

여기서 `p.release()` 는 p의 메모리에 대한 소유권을 해제하고, 메모리를 가르키는 raw 포인터를 반환한다.
(raw pointer: 일반 포인터라고 이해하면 된다.)

조심

`p.release()`를 사용해 메모리에 대한 포인터를 넘겨받으면,
이걸 다른 unique_ptr에게 전달하지 않는 이상 우리의 책임이다.
즉 raw 포인터를 우리가 따로 해제해 주어야 한다.

```
auto p = make_unique<int>(1);
auto* rawPtr = p.release();

// 다 쓴 다음엔 꼭!
delete rawPtr;
```

조심

`p.get()`을 쓰면 unique_ptr가 가지고 있는 메모리에 대한 raw 포인터를 받을 수 있다.
대신 소유권은 그대로이다. 이 포인트를 해제하진 말자! (이건 p의 책임이다.)

```
auto p = make_unique<int>(1);
auto* rawPtr = p.get();

cout << *rawPtr << endl;
```

unique_ptr: copy는 못하지만 return은 해도 괜찮다.

이유는? 다음에.

```
unique_ptr<int> clone(int p) {
    unique_ptr<int> ret(new int(p));
    return ret; //?
}
```

shared_ptr (공유) vs. unique_ptr (소유)

	Heap Memory를 공유	**Heap Memory**를 소유
Parent	shared_ptr	unique_ptr
Child	weak_ptr	T*

최종 예시: **Sudoku**

소스 코드: https://github.com/lasagnaphil/cpp-sudoku

```
template <class T>
using Array2d = std::array<std::array<T, 9>, 9>;

class Board {
public:
    Board(Array2d<int>& data);
    enum class SolveResult { COMPLETE, INCOMPLETE, ERROR };
    ...

private:

    void createGroups();
    void print() const;

    Array2d<std::shared_ptr<Block>> blocks;
    std::array<std::shared_ptr<Group>, 27> groups;
};
```

```
class Block {
public:
    Block(int sol);
    Block(int sol, std::vector<std::shared_ptr<Group>>& groups);
    ...
private:
    Candidates candidates;
    std::vector<std::shared_ptr<Group>> groups;
};
```

```
class Block;

class Group {
public:
    typedef std::vector<std::weak_ptr<Block>>::iterator BlockIt;
    Group(std::vector<std::weak_ptr<Block>>& blocks);
    ...
private:
    std::vector<std::weak_ptr<Block>> blocks;
    Candidates candidates;
};
```

```
class Loader {
public:
    Loader(string filename);
    ...
private:
    std::unordered_map<string, Array2d<int>> sudokuLevels;
    std::unique_ptr<Board> board;
};
```

끄으읕

기말 시험기간 이후에 봅시다 ㅠㅠ