

# 인트아이 C++ 심화 스터디

RAII, SMART POINTER, SCOPE GUARD

# 1회차 Quiz 답안

```
#include "quiz.h"

Person::Person(const std::string &name)
: name(name) {

const std::string &Person::get_name() const {
    return name;
}

Professor::Professor(const std::string &name, const std::string &department_name)
: Person(name), department_name(department_name) {

const std::string &Professor::get_department_name() const {
    return department_name;
}

Student::Student(const std::string &name, int grade)
: Person(name), grade(grade) {

int Student::get_grade() const {
    return grade;
}

Lecture::Lecture(const std::string &name, const Professor &professor, const
std::vector<Student> &students)
: name(name), professor(professor), students(students) {

void Lecture::add_student(const Student &student) {
    students.push_back(student);
}

const std::string &Lecture::get_name() const {
    return name;
}

const Professor &Lecture::get_professor() const {
    return professor;
}

const std::vector<Student> &Lecture::get_students() const {
    return students;
}
```

# RAII

Resource Acquisition Is Initialization

(자원의 획득은 초기화)

(=생성자)

↓ 대우 명제

소멸은 자원의 반납

(=소멸자)

Class의 생성자에서 **new**를 하고, 소멸자에서 **delete**를 한다.

# RAII의 활용

(C++이 어려워지는 이유)

- 일반적으로 C++에서는 모든 것을 프로그래머가 **직접 호출**해야 함  
예) `int *i = new int;`를 했으면, 마지막에 `delete i;`를 해야 함. 안 하면 메모리 누수
- 하지만, **예외적으로 Class의 소멸자**는 코드에서 **자동으로 호출**되는 기능임
- 따라서, **소멸자를 이용**하면 코드에서 특정 부분이 **자동으로 동작**하게 만들 수 있음
- 그렇다면, **delete가 자동**으로 되는 **포인터**를 만들 수 있지 않을까? → **스마트 포인터**

# Smart Pointer

- 프로그래머가 직접 delete하지 않아도, 사용이 끝나면 **자동으로 delete**되는 포인터
- **RAII를 이용해** 구현, **소멸자에 delete**하는 코드를 넣어 줌
- C와 달리, C++ 코드는 스마트 포인터를 이용해 **포인터가 최대한 안 보이게 숨겨야** 함.
  - ✓ 프로그래머가 **직접 관리하는 포인터**가 많아질수록 프로그래머가 **실수할 확률**이 증가
  - ✓ **동적할당**이 필요하면 RAII로 한 번 감싸서 사용
  - ✓ 가능한 한 **스마트 포인터에 의존**해 포인터를 관리

# Smart Pointer 직접 구현

```
#pragma once
```

```
template<typename T>
class Box {
public:
    explicit Box(T *ptr)
        : ptr(ptr) {} 생성자에서 포인터를 받아서 멤버변수에 저장

    T* get() const {
        return ptr;
    }

    ~Box() { 소멸자에서 포인터를 delete
        delete[] ptr;
    }

private:
    T *ptr;
};
```

```
#include <iostream>
#include "box.h"

void test_box() {
    Box<int> i{ new int }; test_box 함수 안에서만 사용가능
    std::cout << i.get() << '\n';

    // int a[100000000]; // error: too large

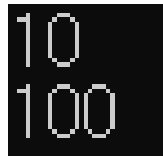
    Box<int> a{ new int[100000000] }; test_box 함수 안에서만 사용가능
    std::cout << a.get() << '\n';
} test_box 가 끝나면 Box 안에 들어있는 포인터는 자동으로 delete

int main() {
    test_box();
    return 0;
}
```

# STL의 Smart Pointer 이용

```
#include <iostream>
#include <memory>

void test_unique_ptr() {
    std::unique_ptr<int> i{ new int(10) };
    std::cout << *i << '\n';
    *i = 100;
    std::cout << *i << '\n';
}
```



10  
100

```
void test_unique_ptr_better() {
    std::unique_ptr<int> i = std::make_unique<int>(10); new보다 std::make_unique가 더 좋음
    std::cout << *i << '\n';
    *i = 100;
    std::cout << *i << '\n';
}
```

[https://en.cppreference.com/w/cpp/memory/unique\\_ptr/unique\\_ptr](https://en.cppreference.com/w/cpp/memory/unique_ptr/unique_ptr)

# Smart Pointer 주의사항 1

- 할당받은 포인터를 함수 외부로 들고나가면 안 됨
  - ✓ 포인터의 수명이 소유자의 수명보다 짧아야 함

```
int *p;
```

```
void test_unique_ptr_bad() {  
    std::unique_ptr<int> i = std::make_unique<int>(10); 함수가 포인터를 소유 중  
    std::cout << *i << '\n';  
    p = i.get(); 포인터를 함수 외부로 가져나감  
}
```

```
int main() {  
    test_unique_ptr_bad();  
    std::cout << *p << '\n'; p는 이미 delete된 포인터  
    return 0;  
}
```



```
10  
-572662307
```



# Smart Pointer 주의사항 2

- 복사 불가, 이름이 **Unique** Pointer인 이유

```
void test_unique_ptr_bad() {  
    std::unique_ptr<int> i = std::make_unique<int>(10);  
    std::cout << *i << '\n';  
    std::unique_ptr<int> i2 = i; // error: cannot copy  
}
```

영어사전



고급 검색

전체 | 단어·속어 | 뜻풀이 | 예문 | 유의어 | 영영사전

T

T

T

ㅁ

3000 ★★

## unique

1. 유일무이한 2. 특별한 3. 고유의

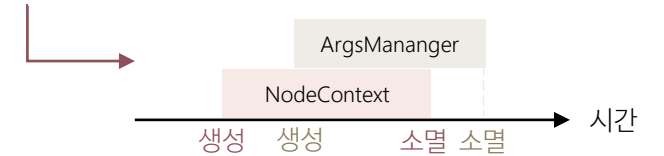
+ 단어장 저장

```

43 struct NodeContext {
44     ///! libbitcoin_kernel context
45     std::unique_ptr<kernel::Context> kernel;
46     ///! Init interface for initializing current process and connecting to other processes.
47     interfaces::Init* init{nullptr};
48     std::unique_ptr<AddrMan> addrman;
49     std::unique_ptr<CConnman> connman;
50     std::unique_ptr<CTxMemPool> mempool;
51     std::unique_ptr<const NetGroupManager> netgroupman;
52     std::unique_ptr<CBlockPolicyEstimator> fee_estimator;
53     std::unique_ptr<PeerManager> peerman;
54     std::unique_ptr<ChainstateManager> chainman;
55     std::unique_ptr<BanMan> banman;
56     ArgsManager* args{nullptr}; // Currently a raw pointer because the memory is not managed by this struct
57     std::unique_ptr<interfaces::Chain> chain; = 메모리가 NodeContext 구조체에서 관리되지 않기 때문에 일반 포인터를 사용
58     ///! List of all chain clients (wallet processes or other client) connected to node.
59     std::vector<std::unique_ptr<interfaces::ChainClient>> chain_clients;
60     ///! Reference to chain client that should used to load or create wallets
61     ///! opened by the gui.
62     interfaces::WalletLoader* wallet_loader{nullptr};
63     std::unique_ptr<CScheduler> scheduler;
64     std::function<void()> rpc_interruption_point = [] {};
65
66     ///! Declare default constructor and destructor that are not inline, so code
67     ///! instantiating the NodeContext struct doesn't need to #include class
68     ///! definitions for all the unique_ptr members.
69     NodeContext();
70     ~NodeContext();
71 };

```

포인터를 unique\_ptr로 표현



ArgsManger의 수명 > NodeContext 수명

unique\_ptr 사용 불가

# Bitcoin 소스코드

# Scope Guard

- 일반적인 경우에서의 RAI의 활용
- 생성자와 소멸자를 이용해서 특정 기능이 **자동으로 동작**하도록 구현

```
#pragma once
#include <iostream>

class CoutGuard {
public:
    CoutGuard() {
        std::cout << "{\n";
    }

    ~CoutGuard() {
        std::cout << "}\n";
    }
};
```

```
#include <iostream>
#include "guard.h"

void test_guard() {
    CoutGuard _;
    std::cout << "after guard\n";
}
```

```
{
after guard
}
```

함수의 시작과 끝에 자동으로 종괄호를 출력함

영어사전

scope



고급 검색

전체 | 단어·속어 | 뜻풀이 | 예문 | 유의어 | 영영사전

T

T

T

ㅁ



## scope

1. 기회 2. 살살이 살피다 3. 범위

+ 단어장 저장

# STL에서의 Guard 활용

조건: i를 사용하기 전에는 mutex.lock을 해야 함, 사용이 끝나면 unlock을 해야 함

```
#include <mutex>

std::mutex mutex;
int i = 10;

void test_mutex() {
    mutex.lock();
    std::cout << "Use i\n";
    i += 1;
    mutex.unlock();
}
```

test\_mutex 함수 시작과 끝에 lock과 unlock을 호출

```
#include <mutex>

std::mutex mutex;
int i = 10;

void test_mutex_guard() {
    std::lock_guard<std::mutex> lock(mutex);
    std::cout << "Use i\n";
    i += 1;
}
```

mutex\_guard를 이용하면 자동으로 lock과 unlock을 호출

# Quiz

1. n번째 피보나치 항을 구하는 fibonacci 함수를 작성하시오. (재귀함수 사용)
2. 주어진 BenchGuard 클래스를 이용해서 25번째 피보나치 항을 계산하는데 걸리는 시간을 출력하시오. `Elapsed time: 2e-07 seconds`

## quiz.h

```
#pragma once
#include <chrono>

class BenchGuard {
public:
    BenchGuard();
    ~BenchGuard();

private:
    std::chrono::time_point<std::chrono::steady_clock> start;
};

unsigned long long fibonacci(int n);
```

## quiz.cpp

```
#include "quiz.h"
#include <chrono>
#include <iostream>

BenchGuard::BenchGuard()
: start(std::chrono::steady_clock::now()) {}

BenchGuard::~BenchGuard() {
    auto end = std::chrono::steady_clock::now();
    std::chrono::duration<double> diff = end - start;
    std::cout << "Elapsed time: " << diff.count() << " seconds\n";
}
```

# Hint

다 풀기 전까지는 되도록이면 페이지를 넘기지 말 것





```
void bench() {  
    BenchGuard _;  
    // 측정할 함수 넣기  
}  
  
int main() {  
    bench();  
    return 0;  
}
```