STL - 3주 2일

Wulong

3주 2일차 강의입니다. STL은 이론 한 시간, 실습 한 시간으로 계획한 강의입니다. 그런데 3주 1일차 강의는 너무 설명만 많이 했습니다. 원래 내 강의는 실습시간을 따로 구분하지 않고 설명과 코딩을 반복하는 스타일이라 계속 말하면서 진행해 나가기는 하지만 글로도 계속 떠들기만 한 기분입니다.

STL 강의에 필요한 기본지식을 복습하고 책 1장을 훑어보는 중입니다. 책 1.6절 '스마트 포인터'와 1.8절 '함수를 인자로 전달하기'를 살펴보겠습니다. 이 항목은 STL을 공부하는 데 있어 꼭 알아야 할 것들입니다. 다시 강조합니다만 공부는 반복입니다. 여러분들이 내 강의가 편안하게 느껴졌다면 강의 내용과 같은 주제를 여러 번 반복하여 학습했기 때문일 것이라고 확신합니다. 아는 내용을 반복하면서 새로운 내용을 추가하며 설명하도록 하겠습니다. 더 기억에 남도록 말이죠. 계속 글로 쓰는 것 보다 이번 강의는 코드를 만들어 보며 알아야 할 것들을 설명해 보겠습니다. 먼저 호출가능 타입부터 가 보겠습니다.

호출가능 타입

호출가능 타입(callable type)이란 무엇일까요? 말 뜻 그대로 입니다. 호출할 수 있는 자료형이라는 말입니다. 호출할 수 있는 자료형까지 붙여 생각해보면 설명하기 어렵겠지만 "호출할 수 있는 것이 뭐지?"까지 끊어 생각해 보면 대답할 수 있을 겁니다. 호출할 수 있는 것은 바로 함수입니다. 우리는 함수를 호출할 수 있습니다. 호출하려고 만든 것이 함수입니다. 여러분 그럼 다음과 같은 질문에 대답해 보세요.

- 함수란 무엇입니까?
- 왜 사용합니까?
- 어떻게 사용합니까?
- 다른 함수와 어떻게 구분합니까?

누군가 이런 질문을 한다면 뭐라고 대답하겠습니까? 아마 같은 질문이라도 대답하는 사람에 따라 서로 다른 답이 나올 수도 있을 것입니다. '**프로그래밍 언어와 함수**'라는 검색어로 검색해 보세요. 지금요. 함수는 코드를 재활용하고 입력에 대한 특정 동작을 수행하고 결과를 되돌려 준다와 같은 설명과 함께 다양한 함수들 그리고 관련된 프로그래밍 언어들과 같은 내용을 볼 수 있을 것입니다. 무슨 소리인지 모르는 설명도 있을 겁니다.

첫 강의에서 이야기 한 것과 같이 찾아볼 수 있는 자료를 내가 다시 옮겨 적을 것은 없습니다. 내가 시간을 들여 강조하고 싶은 것은, 내 강의에서는 어떤 주제에 대하여 생각해 봐야 할 점을 제시하고 내가 알고 있는 내용을 설명하여 학생들이 전체적인 흐름을 이해하도록 노력해 봐야 한다는 점입니다. 이 강의가 모든 것을 다 설명하고 있다고 생각하면 안 됩니다.

나는 그렇게 할 수 없습니다. 그럴 생각도 없습니다. 중요 뼈대만 설명할 뿐입니다. 그것도 상세한 설명을 하려고 하지 않을 것입니다. 관련 내용을 꼭 찾아보고 복습해야 합니다. 진짜 걱정되어 하는 소리입니다. 이 강의를 이해했다고 STL을 안다고 할 수 없습니다. 더 많이 **찾아보고 스스로 공부**해야 합니다.

나는 함수를 이렇게 설명합니다. 함수란 프로그램의 일부분을 떼어 필요할 때 불러 쓸 수 있도록 특별한 곳에 따로 기록해 놓은 것입니다. 불러 쓸 수 있으려면 함수에 이름을 붙여야 합니다. 그래서 함수에는 이름이 있습니다. 대표함수로는 C의 printf나 sin, cos 함수를 생각해 볼 수 있겠습니다. 물론 프로그래머가 직접 부를 일이 없는 람다와 같은 이름 없는 함수도 있습니다. 특별한 곳에 기록해 놓는다고 했는데 함수가 어디에 기록되어 있나 하면 바로 CODE 영역이라는 메모리 영역입니다. 지난 시간에 설명하지 않은 영역이 바로 여기입니다. 이 메모리 영역은 프로그램이 시스템 메모리에 올라가 실행될 때 구분되는 것입니다. 함수의 이름은 CODE 영역에 기록되어 있는 해당 함수의 시작 번지를 의미합니다. 즉 함수의 이름은 번지가 저장되어 있는 포인터라는 말입니다. 간단한 코드를 보겠습니다.

```
#include <iostream>
#include <cmath>
using namespace std;

void f( int n )
{
    cout << "f:" << n << endl;
};

int main()
{
    f( 1 );                         // 일반호출

    (*f)( 2 );                      // 함수이름은 함수의 시작번지를 가리키는 포인터

    void* p = f;                    // 함수의 시작번지를 포인터에 저장

    cout << "함수 f의 주소: " << p << endl;

    int n{ };
    cout << "스택의 주소: " << &n << endl; // 영역이 달라 주소가 많이 차이 날 것임

    (*(void(*)(int))p)(3);          // 주소 p를 함수로 바꿔 인자 3으로 호출
}
```

함수의 이름이 번지이기 때문에 함수 f를 간접참조 연산자(indirection operator) *을 앞에 붙이면 *f는 이론상 함수 그 자체가 되어야 합니다. 그렇지만 함수 그 자체라는 것을 표현하는 것도 함수의 시작번지 말고는 의미가 없습니다. 즉, f나 *f나 같다는 것이죠. 포인터 p를 이용하면 CODE 세그먼트의 번지를 엿볼 수 있습니다. 비교를 위해 STACK에 있는 변수 n의 번지를 같이 출력해 봤습니다.

마지막 줄을 이해할 수 있나요? p에 저장된 주소를 함수 주소로 형 변환(type conversion)한 것입니다. 이 문장에서 살펴봐야 할 것은 p가 지금은 함수 f의 번지를 저장하고 있지만 언제든지 다른 함수의 번지도 저장할 수 있다는 것입니다. p가 가리키는 함수가 변할 수 있다는 말입니다. p를 실행했는데 다른 기능이 실행될 수 있다는 것이죠. 구체적인 예를 조금 후에 들어 보겠습니다.

함수에는 특정한 기능을 구현한 함수, 우리가 매번 프로그램하지 않아도 되는 코드, 우리가 구현하기엔 매우 어려운 함수 등 여러 가지가 있습니다. 그 중 sine 함수를 잠깐 보겠습니다. sine 함수는 c의 math.h에 선언되어 있습니다. 입력으로 radian 값을 전달하면 그에 해당하는 sine 값을 출력합니다. 다음은 표준 함수 sin을 이용하여 화면에 정현파(sine wave)를 출력하는 프로그램입니다. 이 강의록에 있는 모든 코드는 내가 생각하는 것을 바로 바로 코딩한 것이기 때문에 오류가 있을 수 있습니다. 혹시 틀린 곳이 있더라도 여러분이 바로 고칠 수 있으리라 생각합니다. 흥미를 위해 주석은 붙이지 않았습니다. 몇 줄 안 되니 잠깐 쉬어간다 생각하고 돌려 보세요.

```
#include <iostream>
#include <cmath>
#include <thread>
using namespace std;


// sin 곡선을 화면에 출력합니다 - 출력창을 가로 80칸으로 설정하라

int main()
{
    double rad { 2 * 3.1416 };
    int val;
    int freq;

    cout << "주파수를 입력하세요(Hz) 추천값은 10 - ";
    cin >> freq;

    double inc = 1. / freq;

    while (true) {
            val = sin( rad ) * 39 + 39;
            for (int i = 0; i < val; ++i)
                    cout << ' ';
            cout << '*' << endl;
            rad += inc;
            this_thread::sleep_for( 33ms );
    }
}
```

이 프로그램을 만들어 본 것은 코드의 재사용과 함수 이름의 의미를 학생들이 쉽게 이해할

수 있으리라 생각했기 때문입니다. sine 값을 구하기 위해 함수 sin을 호출합니다. 이름이 있으면 여러 번 부를 수 있습니다. 함수를 구분하는 기본은 이름입니다. 함수 이름이 같다면 그 다음은 인자의 갯수와 타입으로 구분합니다. C++ 언어는 같은 이름의 함수도 쓸 수 있습니다. 함수의 인자가 서로 달라 구분가능하다면 말이죠? 이건 어려운 질문이긴 한데 똑같은 이름의 함수를 컴파일러는 어떤 코드를 만들어 낼까요? 찾아보면 재미있는 단어를 발견할 수 있을 겁니다. 발음이 dangling과 비슷한 단어가 나올 거에요. 한 걸음 더 나아가 함수가 클래스의 멤버라면 이름과 인자가 같더라도 한정자(qualifier)에 따라 서로 다른 함수로 구분할 수도 있습니다. 2주 강의에서 operator[ ] 함수에 const 한정자를 붙인 예가 있었습니다.

지금까지 설명은 진짜 복습이었습니다. 어려운 점이 전혀 없었습니다. 그런데 **함수를 이렇게만 사용하나요?** 2주 강의에 프로그램의 기능을 어떻게 바꾸는 지 설명한 것이 기억나나요? 자료 정렬부분에 잘 설명되어 있을 겁니다. 그때 설명한 내용은 어떤 함수가 다른 함수를 인자로 전달받아 자신의 기능을 바꿀 수 있다는 것이었습니다. 한 가지 더 살펴보죠. 이것은 **프로그램이 실행되고 있는 동안에 기능을 바꾸는 내용**입니다. 예를 들어 봅니다. 쿠키런 게임은 단순하게 점프와 슬라이드 버튼만으로 게임을 진행합니다. 그런데 설정에서 두 버튼의 기능을 바꿀 수 있습니다. 이건 제일 단순한 예입니다. 괜찮은 게임은 사용하는 모든 입력 장치의 버튼을 사용자가 설정할 수 있도록 합니다.

다음 코드는 키 입력에 따라 점프와 슬라이드를 화면 출력합니다.

```
#include <iostream>
using namespace std;

void jump();
void slide();

int main()
{
    char key;

    bool flag{ false };

    while (true) {
            cout << "a 나 d 키를 누르세요. 끝내려면 q를 누르세요: ";
            cin >> key;

            switch (key) {
            case 'a':
            case 'A':
                    jump();
                    break;
            case 'd':
            case 'D':
                    slide();
                    break;
            case 'q':
```

```
                case 'Q':
                        flag = true;
                        break;
                default:
                        break;
                }
                if (flag == true)
                        break;
        }
}

void jump()
{
        cout << "점프" << endl;
}

void slide()
{
        cout << "슬라이드" << endl;
}
```

가정해 봅시다. 이렇게 코딩해 놨는데 **팀장**이 버튼의 기능을 바꿀 수 있게 프로그램을 바꾸면 좋겠다고 하는 겁니다. 그것도 당장 말이죠! 이 프로그램의 문제점을 알 수 있다면 그렇게 하는 것이 어렵지 않습니다. 기능변환을 어렵게 하는 이 코드의 문제점을 찾을 수 있나요?

네. 그렇습니다. 이 코드는 유연하지 않네요. 키보드 'a'를 누르면 바로 jump를 호출합니다. 고칠 수 있겠습니까?

[실습] 키보드 'c'를 누르면 점프와 슬라이드 버튼의 기능이 바뀌도록 하라. (10분)

나는 이렇게 해 봤습니다. 바뀐 부분만 적겠습니다.

```
void (*func1)() = jump;
void (*func2)() = slide;

int main()
{
        char key;

        bool flag { false };

        while ( true ) {
                cout << "a 나 d 키를 누르세요. 기능전환은 c, 끝내려면 q를 누르세요: ";
                cin >> key;
```

```
                switch (key) {
                case 'a':
                case 'A':
                        func1();
                        break;
                case 'd':
                case 'D':
                        func2();
                        break;
                case 'c':
                case 'C':
                        {
                                auto temp{ func1 };
                                func1 = func2;
                                func2 = temp;
                        }
                        break;
                default:
                        break;
                }
                if (flag == true)
                        break;
        }
}
```

어떻습니까? 역시 어렵지 않죠? 직접 함수를 호출하는 코드를 다른 곳을 한 번 거쳐 가도록 코딩하는 겁니다. 이 코드를 더 발전시키면 명령 패턴(command patter)이 될 겁니다만 이만큼만 해도 프로그램이 조금 유연해진 것 같습니다. 기억하기로는 책 "게임 프로그래밍 패턴"의 제일 처음 내용이 바로 이것이었던 것 같습니다.

함수를 호출한다는 것은 함수의 시작번지로 CPU의 프로그램 카운터를 옮기는 것입니다. 어셈블리어로는 대부분의 CPU에서 jump 명령어로 이렇게 합니다. 말이 조금 어려웠습니다. **C++ 문법으로 호출한다는 것은 어떤 것에 ( )을 붙인다는 말입니다**. 왜 어떤 것이라고 했는가 하면 그렇게 할 수 있는 것이 하나가 아니기 때문입니다. 위의 코드들에서 보면

- 함수
- 함수 포인터

에 ( )을 붙이고 세미콜론으로 문장을 끝냈거든요. 이 때 ( )을 **함수 호출 연산자**(function call operator)라고 부르고 이것을 붙일 수 있는 타입을 **호출가능 타입**(callable type)이라고 부릅니다. ( ) 연산자를 붙일 수 있는 다른 타입을 말해 볼 수 있습니까? 네, 지난 강의에서 소개한 이름 없는 함수 람다가 있습니다. 람다도 호출가능한 타입입니다. 다음 예를 봐 주세요.

```
#include <iostream>
using namespace std;

int main()
{
    []() {
            cout << "안녕? 난 람다라고 해!" << endl;
    };
}
```

프로그램이 문제없이 실행될 테지만 화면에 출력되는 것은 없죠? 왜 그렇습니까? 호출가능 타입인 람다를 호출하지 않았기 때문입니다. 람다를 호출하고 람다의 타입도 화면 출력해 보겠습니다.

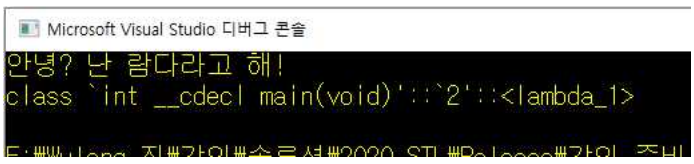
```
#include <iostream>
using namespace std;

int main()
{
    auto callableType = []() {
            cout << "안녕? 난 람다라고 해!" << endl;
    };

    // 호출가능 타입은 ()를 붙일 수 있다.
    callableType();

    cout << typeid(callableType).name() << endl;
}
```

실행화면을 봐 주세요. 람다의 자료형은 읽기 싫게 보이네요.



앞에서 ( )을 함수호출 **연산자**라고 했습니다. 그럼 연산자 오버로딩이 가능하겠네요? 어디에서요? 클래스에서죠.

어떤 클래스가 함수호출 연산자를 오버로딩하고 있다면 그 클래스 객체는 호출가능하며 클래스는 호출가능 타입이 됩니다.

```
#include <iostream>
using namespace std;

class Dog {
public:
    void operator( )( int n ) {
            cout << n << " - 부르셨나요?" << endl;
    }
};


int main()
{
    Dog dog;          // dog는 호출가능 객체이고 Dog는 호출가능 타입이다.

    dog( 2020 );
}
```

호출가능 타입이 더 늘었습니다. 다시 정리해 볼까요?

- 함수
- 함수 포인터
- 람다
- ( ) 연산자를 오버로딩한 클래스

람다가 C++ 언어에 등장하기 전에는 ( ) 연산자를 오버로딩한 클래스가 아주 중요했습니다. 지금은 특별한 용도에만 사용합니다. 대부분 람다로 STL의 알고리즘 함수의 기능을 원하는 대로 조정할 수 있으므로 사용할 일이 많이 줄었습니다. 이것들 말고 더 없을까요? 멤버함수 포인터도 호출가능 타입입니다. 또한 bind 템플릿 클래스가 만들어 내는 타입도 호출가능 타입입니다. 이것들은 따로 설명하지 않겠습니다.

이제 질문해 보겠습니다.

C++ 언어의 호출가능 타입은 몇 가지나 있습니까?

관찰력이 있는 학생은 람다의 타입을 출력한 화면에서 답을 알 수 있었을 것입니다. 각 람다는 다른 람다와 구분되는 유일한 타입이라는 것을요. 같은 이유로 ( )을 오버로딩한 클래스도 각자 다른 것들과 구분되는 유일한 타입입니다. 그러니 답은 C++ 언어에는 무한개의 호출가능 타입이 있다고 하겠습니다. 여기에서 또 의문이 생깁니다. **수 없이 많은 호출가능 타입을 통일된 방식으로 표현할 수는 없을까**? function 템플릿으로 그렇게 할 수 있습니다.

function 템플릿은 function 타입을 만들어 내는 데, 호출인자와 리턴 타입으로 호출가능 타입을 표현합니다. 코드를 보고 실행해 보는 것이 이해가 빠를 것입니다.

```
#include <iostream>
#include <functional>
using namespace std;

void f( int n )
{
    cout << n << " - 진짜 그냥 함수" << endl;
}

class Dog {
public:
    void operator()( int n ) {
            cout << n << " - function object" << endl;
    }
};

void(*fp)(int) = f;                         // 함수 포인터

auto lambda = []( int n ) {
    cout << n << " - 이름 없는 함수 람다" << endl;
};

int main()
{
    function<void( int )> func;

    cout << typeid(func).name() << endl;          // func는 function 객체이다

    func = f;
    func( 1 );

    func = Dog();
    func( 2 );

    func = fp;
    func( 3 );

    func = lambda;
    func( 4 );
}
```

인자와 리턴 값의 형식이 같다면 func 객체는 어떤 호출가능 타입이라도 수용할 수 있습니다. 어디에 사용하면 좋을 지 생각해 보세요. 알아두면 앞으로 쓸 데가 꼭 있을 것입니다. 책 54쪽 1.8절을 꼭 읽어보세요. 내 간단한 설명보다 더 자세하고 정확하게 설명하고 있어서 함수를 인자로 전달하는 것이 STL에서 얼마나 중요한지 알 수 있을 것입니다. 내 강의만 보고

책을 안 볼까봐 걱정돼서 다시 말해 봅니다. 꼭 책을 보세요. 책이 없으면 관련 내용을 검색하세요.

- 함수란 무엇입니까?
- 왜 사용합니까?
- 어떻게 사용합니까?
- 다른 함수와 어떻게 구분합니까?

함수에 했던 이 질문에 이제 대답할 수 있을 것입니다. 그럼 질문 하나를 추가하겠습니다. 생각해 보세요.

- 왜 여러 가지 호출가능 타입이 있는 겁니까?

스마트 포인터

포인터는 POD의 일종입니다. 스마트 포인터가 아닌 그냥 포인터는 **raw point**라고 부르기도 합니다. 포인터가 스마트하다는 것은 어떤 의미입니까? 무엇이 다른 것일까요. 스마트하다는 것은 단어 뜻 그대로 그냥 포인터보다 뭔가 기능을 더 할 수 있다는 것을 의미합니다. 어떤 기능을 더 하게 할지는 만들 사람 마음입니다. C++ 표준에는 몇 가지 스마트 포인터가 있습니다. 그 중 자원을 독점하는 **unique_ptr**만 설명하겠습니다. 다른 포인터는 사용이 까다로우며 사용할 일이 거의 없습니다. 물론 책에는 다 잘 설명되어 있습니다. 책 42쪽 1.6절 스마트 포인터를 읽어주세요.

나는 이렇게 설명 하겠습니다. 포인터는 주소를 저장할 수 있는 메모리 공간이고 여기에 적힌 값(메모리 주소)에 * 연산자를 이용하여 주소가 가리키는 메모리에 억세스할 수 있습니다. 메모리에 직접 억세스하는 모든 다른 타입과는 다른 점입니다. **포인터는 자유롭습니다**. 프로그래머는 포인터에 저장된 값을 바꿔 써서 얼마든지 원하는 메모리에 접근할 수 있습니다. 이것이 포인터입니다.

그런데 포인터에 저장한 주소가 new로 할당받은 메모리라면 꼭 신경 써야 할 것이 있습니다. 그것은 바로, 메모리 사용이 끝난 후 반드시 delete로 반환해야 한다는 것을 잊어서는 안 된다는 것입니다. 책을 살펴봤다면 new로 할당받은 공간을 자유 공간(free store)이라고 한다는 것도 알았을 것입니다. C++ 표준은 지난 시간에 정리한 메모리 공간을 조금 다르게 세분하여 부르고 있습니다만 다른 프로그래머들과의 소통엔 그대로 말하는 것이 더 좋다고 생각합니다. new로 자유 공간에서 자원을 확보하는 프로그램을 예로 들어 봅니다.

```
#include <iostream>
using namespace std;
```

```
class Dog {
    int n;

public:
// 생성과 소멸 관찰용
//  Dog() { cout << "생성" << endl; }
//  ~Dog() { cout << "소멸" << endl; }

    void show() {
            cout << "Dog입니다만" << endl;
    }
};

void f( Dog* );

int main()
{
    Dog* p = new Dog;                    // 자원을 확보한다

    p->show();                           // 자원을 사용한다

    f( p );                              // 자원을 사용한다

    delete p;                            // 자원을 반환한다
}

void f( Dog* pDog )
{
    pDog->show();

    delete pDog;
}
```

자원을 확보하는 프로그램의 절차는 다음과 같습니다.

- new로 확보한 자원을 포인터로 저장한다.
- 포인터를 이용하여 자원에 access 한다.
- 사용이 끝나면 delete로 반환한다.

이와 같은 프로그램에서의 문제는 자원을 확보한 시점과 자원을 해제한 시점이 다르다는 것입니다. 경우에 따라서는 위의 예와 같이 다른 곳에서 자원을 이미 반환해 버릴 수도 있습니다. 예를 들려고 만든 이 코드에선 한 번 생성한 객체를 두 번 반환하여 프로그램이 비정상 종료될 수 있습니다. 설마 이렇게 코딩하랴 생각할 수도 있지만 제일 흔한 실수이며 찾아내기 힘든 오류가 바로 이것입니다. 어디선가 new로 만든 객체를 해제하지 않거나 여러 번 해제하거나 하게 되면 쓸 수 있는 메모리가 슬슬 줄어들기 시작합니다. 실제 프로그램은 이와 같이 간단하지도 않을뿐더러 하나의 소스 파일안에 new와 짝이 되는 delete가 있는 경우도 그리 많

지 않습니다.

자원을 독점하는 스마트 포인터인 **unique_ptr**가 바로 이런 상황을 해결하려고 고안된 것입니다. 클래스로 만든 unique_ptr는 자신이 생성될 때 생성자의 인자로 확보할 자원을 받습니다. 이 자원을 멤버 변수로 저장해 놓고 unique_ptr 자신이 소멸될 때 저장한 자원의 소멸자를 호출합니다. 프로그램으로 확인해 보겠습니다.

```
#include <iostream>
#include <memory>
using namespace std;

int main()
{
    unique_ptr<Dog> p { new Dog };          // 자원을 확보한다

    p->show();                              // 자원을 사용한다

    cout << sizeof( p ) << endl;            // unique_ptr의 크기
}
```

포인터 p를 생성할 때 확보한 자원을 인자로 받습니다. 사용법은 일반 포인터와 같습니다. 포인터 p는 지역변수이므로 지역을 벗어날 때 STACK에서 제거됩니다. 이때 p는 Dog의 소멸자를 호출하고 객체를 반환합니다. 프로그래머는 언제 명시적으로 delete를 호출해야 하는가를 고민하지 않아도 됩니다. 포인터 p의 크기는 POD 포인터와 같습니다. 메모리 자원을 더 차지하지 않고 delete를 잊어도 되는 스마트 포인터를 사용하지 않을 이유는 전혀 없습니다. 일반 포인터 대신 unique_ptr을 사용하기를 적극 권장합니다.

unique_ptr는 자원을 독점하기 때문에 p를 복사할 수는 없습니다. 그래서 이 앞의 프로그램에서 f를 호출할 때는 이동문법을 사용하여 소유권을 넘겨야 합니다. 다음과 같이 프로그램합니다.

```
#include <iostream>
#include <memory>
using namespace std;

// class Dog 생략

void f( unique_ptr<Dog> );

int main()
{
    unique_ptr<Dog> p{ new Dog };           // 자원을 확보한다

    p->show();                              // 자원을 사용한다
```

```
    f( move(p) );                                   // 자원을 사용한다
}

void f( unique_ptr<Dog> pDog )
{
    pDog->show();
    cout << "f" << endl;
}
```

객체의 생성과 소멸이 같은 횟수로 실행됨을 확인할 수 있습니다. 코딩해보고 이해한 후 적극적으로 사용합시다.

앞으로

```
    int* p = new int;
```

와 같은 문장은 쓰지 말도록 해 봅시다. 어떻게 쓰라구요?

```
    unique_ptr<int> p { new int };
```

이렇게 써야 하겠습니다.

마지막으로 한 가지 더 이야기하고 이번 강의를 마치겠습니다. **new로 자원을 확보하는 것은 2가지 형태가 있습니다.** 그냥 포인터를 사용할 때는 이렇게 했습니다.

```
int main()
{
    // 객체 한 개
    Dog* dog = new Dog;
    delete dog;

    // 객체 여러 개
    Dog* dogs = new Dog[3];
    delete[ ] dogs;
}
```

생성과 소멸 갯수가 맞죠? unique_ptr로 바꿔 보겠습니다.

```
// 해결해야 할 코드

int main()
{
    // 관찰이 쉽게 코드를 지역으로 나눴다

    {
            // 객체 한 개 - 이렇게 unique_ptr를 생성하는 것이 더 좋다
            auto dog = make_unique<Dog>();
    }

    // 객체 여러 개
    unique_ptr<Dog> dogs{ new Dog[3] };
}
```

어떻습니까? 문제가 무엇인지 확실하게 알 수 있죠? 이 문제를 과제로 내겠습니다.

[3주 2일 - 과제]

[문제 1] "해결해야 할 코드"의 문제점이 무엇인지 적어라.
[문제 2] "해결해야 할 코드"의 문제를 해결하는 프로그램을 작성하라.

문제가 무엇인지 발견하였다면 검색으로 답을 찾을 수 있다.
코드를 완성해 보자.

3주 1일과 3주 2일 과제를
문서 파일 하나로 만들어 (형식은 윈도우 텍스트 문서 형식으로)
4주 1일 강의 시작 전까지 e-class로 제출해 주세요.
과제는 **친구들과 의논하여 해결하는 것을 권장**합니다.