14. 레퍼런스 카운트와 상속구조

Object.h

```
class CObject
{
protected:
        CObject();
        virtual ~CObject();

protected:
        int                     m_iRef;

public:
        void AddRef()
        {
                ++m_iRef;
        }

        int Release()
        {
                --m_iRef;

                if( m_iRef == 0 )
                {
                        delete this;
                        return 0;
                }
                return m_iRef;
        }
```

예를 들어 플레이어 피격 UI가 있다고 해보자. UI 호출시 오브젝트들을 하나 하나 검사하는 것보다 플레이어 객체가 UI를 들고 있고 피격시 해당 UI를 접근/호출 하는 것이 효율적이다.
그런데, 해당 피격 UI가 소멸되어 사라졌을 경우 플레이어는 이 사실을 모르고 계속 해당 메모리 공간을 참조하게 되어 문제가 발생한다.

객체하나를 여러 객체들이 공유할때 발생하는 객체 공유의 문제.

→ 객체들에게 자신을 참조하는 횟수를 알고있게 함.
이를 레퍼런스 카운트 라고 한다. * shared_ptr에서 사용

```
template<typename T>
void Safe_Release_VecList( T& p )
{
        typename T::iterator iter;
        typename T::iterator iterEnd = p.end();

        for( iter = p.begin(); iter != iterEnd; ++iter )
        {
                SAFE_RELEASE( (*iter) );
        }

        p.clear();
}
```

release 호출

```
CScene::~CScene()
{
        Safe_Delete_VecList( m_LayerList );
}
```

소멸자를 호출하는 것이 아닌 레퍼런스 카운트를 줄이는 release를 호출해주고, 오브젝트 객체가 자기자신의 ref count가 0이 되면 소멸자를 스스로 호출한다.
이렇게 하면 소멸자의 접근 지정자에 구애받지 않게 된다.

• 상속구조

Core.h

```
class CCore
{
private:
        static CCore*   m_pInst;

public:
        static CCore* GetInst()
        {
                if( !m_pInst )
                        m_pInst = new CCore;
                return m_pInst;
        }

        static void DestroyInst()
        {
                SAFE_DELETE( m_pInst );
        }

private:
        CCore();
        ~CCore();

private:
        static bool m_bLoop;

private:
        HINSTANCE       m_hInst;
        HWND            m_hWnd;
        HDC                     m_hDC;
        RESOLUTION  m_tRs;

public:
        bool Init( HINSTANCE hInst );
        int Run();

private:
        void Logic();
        void Input( float fDeltaTime );
        int Update( float fDeltaTime );
        int LateUpdate( float fDeltaTime );
        void Collision( float fDeltaTime );
        void Render( float fDeltaTime );
```

cpp

```
bool CCore::Init( HINSTANCE hInst )
{
    m_hInst = hInst;

    // 윈도우 클래스 등록
    MyRegisterClass();

    // 해상도 설정
    m_tRs.iW = 1280;
    m_tRs.iH = 720;

    // 윈도우 창 생성
    Create();

    // 화면 DC 만들어주기
    m_hDC = GetDC( m_hWnd );

    // 타이머 초기화
    if( !GET_SINGLE( CTimer )->Init() )
    {
        return false;
    }

    // 장면 관리자 초기화
    if( !GET_SINGLE( CSceneManager )->Init() )
    {
        return false;
    }

    return true;
```

최상위 관리자 Core가 hdc 만들어 씬 매니저의 함수들을 호출.

```
class CSceneManager
{
private:
        class CScene* m_pScene;
        class CScene* m_pNextScene;

public:
        bool Init();
        void Input( float fDeltaTime );
        int Update( float fDeltaTime );
        int LateUpdate( float fDeltaTime );
        void Collision( float fDeltaTime );
        void Render( HDC hDC, float fDeltaTime );
```

같은 함수를 계속 해서 호출해준다.

↓ 씬 호출

↓ 레이어 호출

↓ 레이어의 각 오브젝트 호출