

shared_ptr, weak_ptr 작동 방식 내부

유영천

<https://megayuchi.com>

Tw: @dgtman

사전지식

- weak_ptr, shared_ptr 모두 _Ptr_base를 상속합니다.
- _Ptr_base의 _Rep 멤버는 _Ref_count_base의 포인터를 저장합니다. 이것이 ref count를 저장하는 컨트롤 블록입니다.
- _Ref_count_base의 구조는 다음과 같습니다.
 - vtable entry ptr(4bytes) | _Atomic_counter_t _Uses(4 bytes) | _Atomic_counter_t _Weaks (4bytes) 로 12 bytes입니다.

테스트를 위한 class 선언은 다음과 같습니다. 메모리 watch 창에서 확인하기 쉽도록 클래스 선언에서 문자열을 집어넣었습니다.

sizeof(CParent) 해보면 24바이트가 나옵니다.

std::weak_ptr<CChild> m_child의 사이즈 8 bytes + 문자열 버퍼 m_str의 사이즈 16 bytes = 24 bytes입니다.

```
class CParent
{
public:
    std::weak_ptr<CChild> m_child;

    char m_str[16] = { "CParent" };
    ~CParent();
};
```

이 코드로 테스트를 진행합니다.

결론부터 얘기하면 메모리 해제는 CParent의 실제 메모리는 weakParent.reset()이 호출되고 나서야 해제됩니다. 그 과정을 확인하겠습니다.

```
void TestSharedPtrWithNoMember()
{
    std::shared_ptr<CParent> parent = std::make_shared<CParent>(); // CParent의 use_ref = 1, weak_ref = 1
    std::shared_ptr<CChild> child = std::make_shared<CChild>();    // CChild의 use_ref = 1, weak_ref = 1

    std::weak_ptr<CParent> weakParent = parent; // CParent의 use_ref = 1, weak_ref = 2
    std::weak_ptr<CChild> weakChild = child;    // CChild의 use_ref = 1, weak_ref = 2

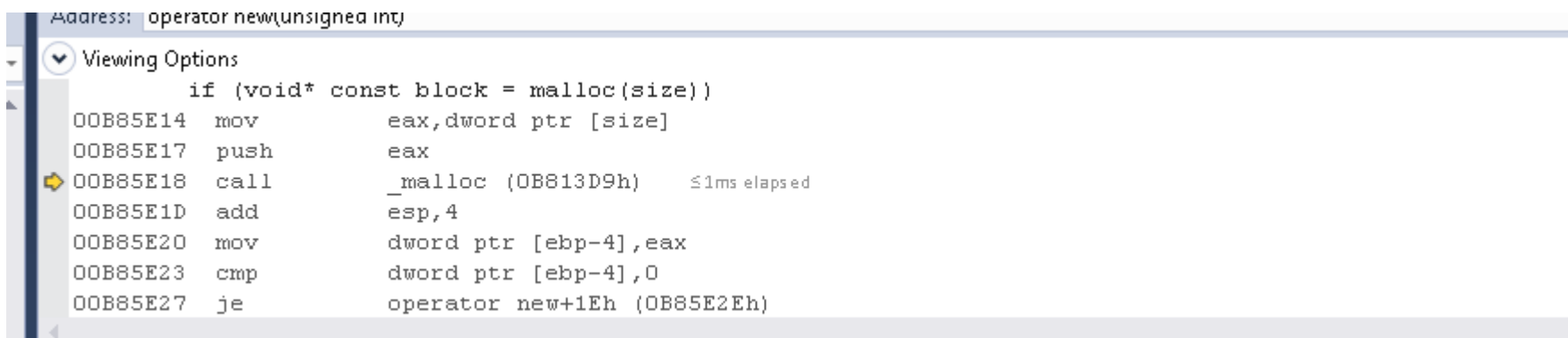
    parent = nullptr; // CParent의 use_ref = 0
                      // use_ref == 0이므로 _Destroy() 호출 -> CParent의 소멸자 호출. 멤버 m_child가 empty이므로 CChild메모리에 영향 없음.
                      // CParent의 use_ref == 0 && weak_ref == 1이므로 메모리 해제 안됨.

    child = nullptr; // CChild의 use_ref = 0
                    // use_ref == 0이므로 _Destroy() 호출 -> CChild의 소멸자 호출. 멤버 m_parent가 empty이므로 CParent메모리에 영향 없음.
                    // CChild의 use_ref == 0 && weak_ref == 1이므로 메모리 해제 안됨.

    weakParent.reset(); // <- CParent의 weak_ref = 0, use_ref == 0 && weak_ref == 0이므로 CParent의 실제 메모리는 이 시점에서 해제됨.
    weakChild.reset();  // <- CChild의 weak_ref = 0, use_ref == 0 && weak_ref == 0이므로 CChild의 실제 메모리는 이 시점에서 해제됨.
}
```

```
std::shared_ptr<CParent> parent = std::make_shared<CParent>(); // CParent의 use_ref = 1, weak_ref = 1
```

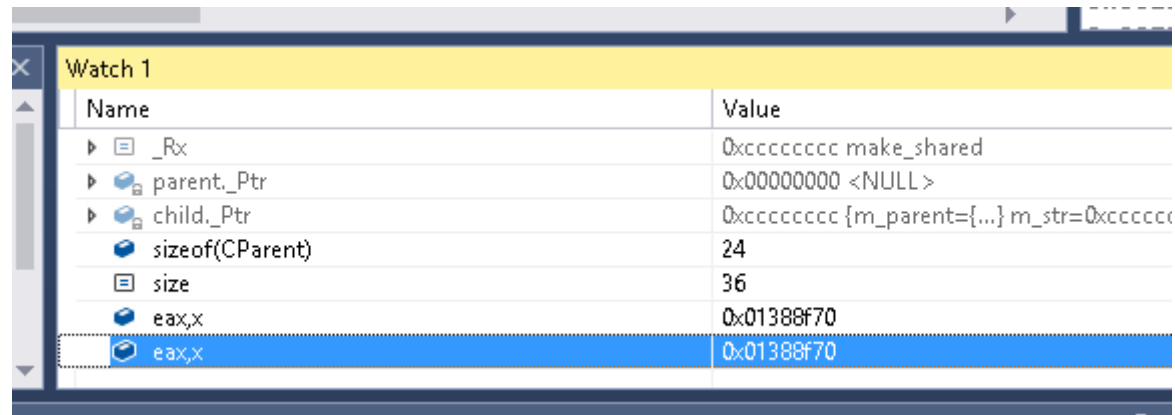
make_shared<CParent>()를 호출하면 내부적으로 메모리를 할당하는데 이 코드를 따라가면 최종적으로 malloc()을 호출합니다.
malloc에 전달되는 사이즈는 36입니다.



```
Address: operator new(unsigned int)
Viewing Options
if (void* const block = malloc(size))
00B85E14 mov     eax,dword ptr [size]
00B85E17 push    eax
00B85E18 call    _malloc (0B813D9h)    51ms elapsed
00B85E1D add     esp,4
00B85E20 mov     dword ptr [ebp-4],eax
00B85E23 cmp     dword ptr [ebp-4],0
00B85E27 je     operator new+1Eh (0B85E2Eh)
```

CParent의 사이즈 24 bytes + 컨트롤 블록의 사이즈 12 bytes = 36 bytes입니다.
이렇게 객체의 메모리와 컨트롤 블록의 메모리는 한번에 할당됩니다.

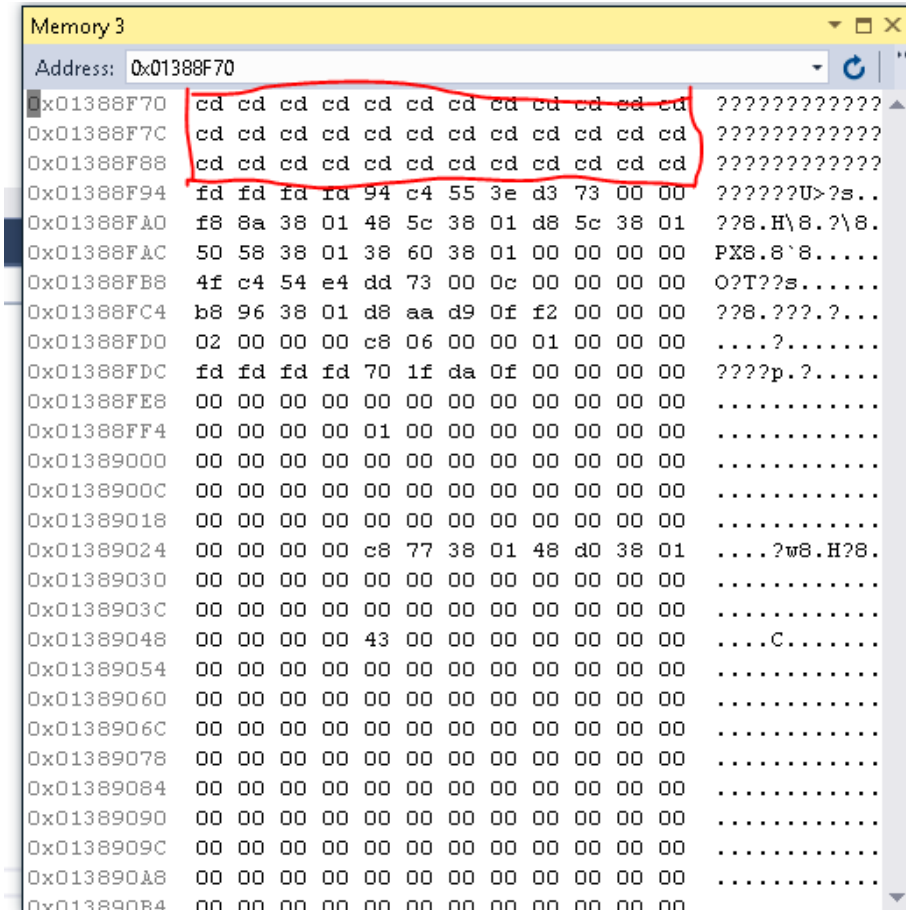
malloc호출후 리턴된 포인터를 확인하면 0x01388f70입니다.
make_shared<CParent>()호출후 이 주소가 parent._Rep에 저장됩니다.
parent._Ptr에는 0x01388f70 + 12 bytes 주소가 저장됩니다.



The screenshot shows a debugger's 'Watch 1' window. It contains a table with two columns: 'Name' and 'Value'. The table lists several variables and their current values. The variable 'eax,x' is highlighted in blue, showing the address 0x01388f70.

Name	Value
▸ _Rx	0xc0000000 make_shared
▸ parent._Ptr	0x00000000 <NULL>
▸ child._Ptr	0xc0000000 {m_parent={...} m_str=0xc0000000
sizeof(CParent)	24
size	36
eax,x	0x01388f70
eax,x	0x01388f70

make_shared()-> malloc()을 통해 얻은 포인터(parent._Rep) 0x01388f70의 메모리를 확인해봅니다.
방금 힙으로부터 할당된 메모리입니다. Debug 런타임이 0xcdcdcdcd로 초기화한 걸 확인할 수 있습니다.

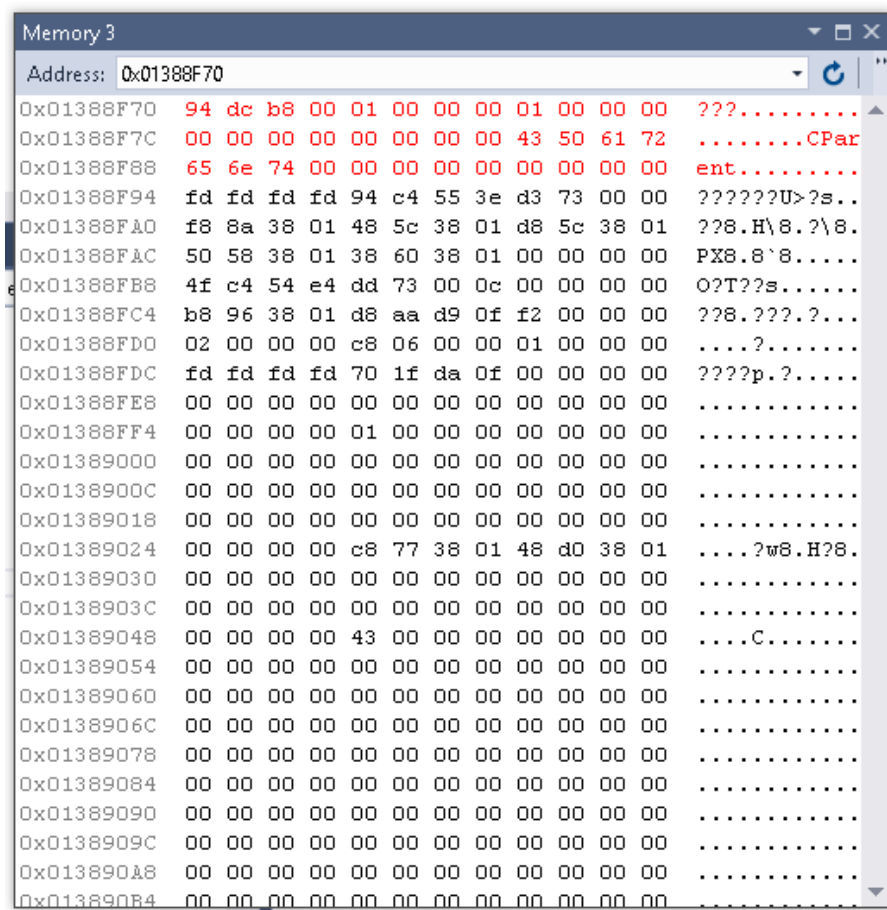


코드를 조금 더 진행시켜서 CParent()의 생성자까지 수행시키고 나서 메모리를 확인합니다.

맨 처음 4 bytes는 vtable, 그 다음 4bytes 는 use ref count = 1 그 다음 4bytes 는weak ref count = 1입니다.

그 다음 8 bytes는 `std::weak_ptr<CChild> m_child` 변수입니다.

그 다음 16 bytes는 문자열입니다. 선언시에 "CParent"를 넣어줬었죠.



도입부에 언급했듯이 weak ref값이 0이 되고나서야 메모리가 해제됩니다.
메모리를 해제될수 있도록 reset()을 호출합니다.

```
void TestSharedPtrWithNoMember()
{
    std::shared_ptr<CParent> parent = std::make_shared<CParent>(); // CParent의 use_ref = 1, weak_ref = 1
    std::shared_ptr<CChild> child = std::make_shared<CChild>(); // CChild의 use_ref = 1, weak_ref = 1

    std::weak_ptr<CParent> weakParent = parent; // CParent의 use_ref = 1, weak_ref = 2
    std::weak_ptr<CChild> weakChild = child; // CChild의 use_ref = 1, weak_ref = 2

    parent = nullptr; // CParent의 use_ref = 0
                        // use_ref == 0이므로 _Destroy() 호출 -> CParent의 소멸자 호출. 멤버 m_child가 empty이므로 CChild메모리에 영향 없음.
                        // CParent의 use_ref == 0 && weak_ref == 1이므로 메모리 해제 안됨.

    child = nullptr; // CChild의 use_ref = 0
                     // use_ref == 0이므로 _Destroy() 호출 -> CChild의 소멸자 호출. 멤버 m_parent가 empty이므로 CParent메모리에 영향 없음.
                     // CChild의 use_ref == 0 && weak_ref == 1이므로 메모리 해제 안됨.

    weakParent.reset(); // <- CParent의 weak_ref = 0, use_ref == 0 && weak_ref == 0이므로 CParent의 실제 메모리는 이 시점에서 해제됨.
    weakChild.reset(); // <- CChild의 weak_ref = 0, use_ref == 0 && weak_ref == 0이므로 CChild의 실제 메모리는 이 시점에서 해제됨.
}
```

malloc호출후 리턴된 포인터이자 parent._Rep에 저장되었던 컨트롤 블록의 포인터 0x01388f70를 확인해봅니다.

Memory 3	
Address:	0x01388F70
0x01388F70	dd dd dd dd dd dd dd dd dd dd dd ??????????
0x01388F7C	dd dd dd dd dd dd dd dd dd dd dd ??????????
0x01388F88	dd dd dd dd dd dd dd dd dd dd dd ??????????
0x01388F94	dd dd dd dd 04 00 00 04 d3 73 00 00 ????...?s..
0x01388FA0	f8 8a 38 01 48 5c 38 01 d8 5c 38 01 ???.H\8.?\\8.
0x01388FAC	50 58 38 01 38 60 38 01 00 00 00 00 PX8.8`8.....
0x01388FB8	4f c4 54 e4 d7 73 00 0c 00 00 00 00 O??T??s.....
0x01388FC4	b8 96 38 01 d8 aa d9 0f f2 00 00 00 ??.?.?....
0x01388FD0	02 00 00 00 c8 06 00 00 01 00 00 00?.....
0x01388FDC	fd fd fd fd 70 1f da 0f 00 00 00 00 ???p.?....
0x01388FE8	00 00 00 00 00 00 00 00 00 00 00
0x01388FF4	00 00 00 00 01 00 00 00 00 00 00
0x01389000	00 00 00 00 00 00 00 00 00 00 00
0x0138900C	00 00 00 00 00 00 00 00 00 00 00
0x01389018	00 00 00 00 00 00 00 00 00 00 00
0x01389024	00 00 00 00 c8 77 38 01 48 d0 38 01?w8.H?8.
0x01389030	00 00 00 00 00 00 00 00 00 00 00
0x0138903C	00 00 00 00 00 00 00 00 00 00 00
0x01389048	00 00 00 00 43 00 00 00 00 00 00C.....
0x01389054	00 00 00 00 00 00 00 00 00 00 00
0x01389060	00 00 00 00 00 00 00 00 00 00 00
0x0138906C	f8 00 00 00 00 00 00 00 00 00 00
0x01389078	00 00 00 00 00 00 00 00 00 00 00
0x01389084	00 00 00 00 00 00 00 00 00 00 00
0x01389090	00 00 00 00 00 00 00 00 00 00 00
0x0138909C	00 00 00 00 00 00 00 00 00 00 00
0x013890A8	00 00 00 00 00 00 00 00 00 00 00
0x013890B4	00 00 00 00 00 00 00 00 00 00 00

결론

- ref count가 저장되는 컨트롤 블록은
 - vtable (4 bytes) | use ref (4bytes) | weak ref (4 bytes) = 12 bytes로 구성됩니다.
- ref count가 저장되는 컨트롤 블록과 객체의 인스턴스 메모리는 따로따로 할당되지 않고 한번에 할당 됩니다.
- use ref(strong ref)가 0이 되면 소멸자는 호출됩니다.
- use ref와 weak ref모두 0이 되어야 메모리가 해제됩니다.