# Smart Pointer

# Smart Pointer

> *기본적으로 힙 영역에 동적 할당된 메모리를 해제하기 위해서는 delete 키워드를 쓰면 된다.*
> *다만, 스마트 포인터를 이용하면 메모리 누수를 더 효과적으로 방지 할 수 있어, 컴퓨터 시스템의 안정성을 높일 수 있다.*

## 1. 정의

- Memory Leak 방어
- 포인터처럼 동작하는 class template

## 2. 사용 방법

- new 키워드를 이용해서 기본 포인터가 특정한 메모리 주소를 가리키도록 초기화
- 스마트포인터에 해당 포인터를 넣어서 사용
  - 이렇게 정의된 스마트 포인터는 수명을 다 했을 때 소멸자에서 delete 키워드를 이용해, 할당된 메모리들을 자동으로 해제하는 기능을 수행

---

# Smart pointer Object

- unique_ptr : 하나의 스마트 포인터가 특정한 객체를 처리할 수 있도록 함
- shared_ptr : 특정한 객체를 참조하는 스마트 포인터가 총 몇개인지를 참조
- weak_ptr : 하나 이상의 shared_ptr인스턴스가 소유하는 객체에 대한 접근 제공
  - 부가적인 역활을 수행한다.

## 1. unique_ptr

- 하나의 스마트 포인터만이 특정한 객체를 처리하도록 할 때 사용
- 특정한 객체의 소유권을 가지고 있을 때만 소멸자가 객체를 삭제 가능

1) Ex

```
// 1. Code & result
#include <iostream>
```

```
using namespace std;
int main(void)
{
        unique_ptr<int> p1(new int(10));
        unique_ptr<int> p2;
        cout << "smart pointer 1 : " << p1 << '\n';
        cout << "smart pointer 2 : " << p2 << '\n';

        cout << "--- 소유권 이전 ---\n";
        p2 = move(p1);
        cout << "smart pointer 1 : " << p1 << '\n';
        cout << "smart pointer 2 : " << p2 << '\n';

        cout << "--- 메모리 할당 해제 ---\n";
        p2.reset();
        cout << "smart pointer 1 : " << p1 << '\n';
        cout << "smart pointer 2 : " << p2 << '\n';

        system("pause");
        return 0;
}
////////////////////////////////////////////////
smart pointer 1 : 012ACFA0
smart pointer 2 : 00000000
--- 소유권 이전 ---
smart pointer 1 : 00000000
smart pointer 2 : 012ACFA0
--- 메모리 할당 해제 ---
smart pointer 1 : 00000000
smart pointer 2 : 00000000
```

2) Ex

```
#include <iostream>
using namespace std;
int main(void)
{
	int* arr = new int[10];
	unique_ptr<int> p1(arr);

	for (int i = 0; i < 10; ++i)
	{
		arr[i] = i;
	}

	for (int i = 0; i < 10; ++i)
	{
		cout << arr[i] << ' ';
	}
	cout << '\n';

	p1.reset();
	for (int i = 0; i < 10; ++i)
	{
		cout << arr[i] << ' ';
	}
	cout << '\n';

	system("pause");
	return 0;
}
//////////////////////////////////////////////////////////////
```

```
0 1 2 3 4 5 6 7 8 9
-572662307 -572662307 -572662307 -572662307 -572662307 -572662307 -572662307 -572662307 -572662307 -572662307
```

## 2. shared_ptr

- 하나의 특정한 객체를 참조하는 스마트 포인터의 개수가 몇 개인지 참조
- 특정 객체를 새로운 스마트 포인터가 참조할 때마다 참조 횟수(Reference Count)가 1씩 증가하며, 각 스마트 포인터의 수명이 다 할 때마다 1씩 감소
- 결과적으로 참조 횟수가 0이 되며 delete 키워드를 이용해 메모리에서 데이터를 자동으로 할당 해제

**1) Ex1**

```
#include <iostream>

using namespace std;

int main(void)
{
        int* arr = new int[10];
        arr[7] = 100;

        shared_ptr<int> p1(arr);
        cout << p1.use_count() << '\n';
        shared_ptr<int> p2(p1);
        cout << p1.use_count() << '\n';
        shared_ptr<int> p3 = p2;
        cout << p1.use_count() << '\n';
        cout << '\n';

        p1.reset();
        cout << "P1 : " << p1.use_count() << '\n';
        cout << "P2 : " << p2.use_count() << '\n';
        cout << "P3 : " << p3.use_count() << '\n';
```

```
        cout << '\n';

        p2.reset();
        cout << "P1 : " << p1.use_count() << '\n';
        cout << "P2 : " << p2.use_count() << '\n';
        cout << "P3 : " << p3.use_count() << '\n';
        cout << '\n';

        cout << "arr[7] : " << arr[7] << '\n';
        cout << '\n';

        p3.reset();
        cout << "P1 : " << p1.use_count() << '\n';
        cout << "P2 : " << p2.use_count() << '\n';
        cout << "P3 : " << p3.use_count() << '\n';
        cout << '\n';

        cout << "arr[7] : " << arr[7] << '\n';

        system("pause");
        return 0;
}
////////////////////////////////////////////////
1
2
3

P1 : 0
P2 : 2
P3 : 2
```

```
P1 : 0
P2 : 0
P3 : 1

arr[7] : 100

P1 : 0
P2 : 0
P3 : 0

arr[7] : -572662307
```

## 3. waek_ptr

- 하나 이상의 shared_ptr 객체가 참조하고 있는 객체에 접근 가능
  - 하지만 해당 객체의 소유자의 수에는 미 포함
- 일반적으로 서로가 상대방을 가리키는 두개의 shared_ptr이 있다면,
- 참조 횟수는 0이 될 수 없기 때문에 메모리에서 해제 될 수 없다.
- weak_ptr은 이러한 순환 참조(Circular Reference) 현상을 제거하기 위한 목적

**1) Ex1**

```
#include <iostream>

using namespace std;

int main(void)
{
        int *arr = new int(1);
        shared_ptr<int> sp1(arr);
        weak_ptr<int> wp = sp1;

        cout << sp1.use_count() << '\n';
```

```
        cout << wp.use_count() << '\n';
        cout << '\n';

        if (1)
        {
                shared_ptr<int> sp2 = wp.lock(); //shared_ptr
포인터 반환

                cout << sp1.use_count() << '\n';
                cout << wp.use_count() << '\n';
                cout << '\n';
        }
        // Scope를 벗어나므로 sp2가 자동 해제 된다.

        cout << sp1.use_count() << '\n';
        cout << wp.use_count() << '\n';
        cout << '\n';

        system("pause");
        return 0;
}
////////////////////////////////////////////////
1
1

2
2

1
1
```