스마트 포인터

항목 6-1

smart pointer #1

학습 목표

- smart pointer 기본 개념
- shared_ptr
- make_shared
- enable_shared_from_this

1. C++ 표준 스마트 포인터

▌ 핵심 개념

C++ 표준에는 다음과 같은 스마트 포인터를 제공합니다.

name		description
auto_ptr		소유권 이전 방식의 스마트 포인터 (deprecated in C++17)
unique_ptr	C++11	독점적 소유권 방식
shared_ptr	C++11	소유권 공유 방식
weak_ptr	C++11	소유권에 참여하지 않음
observer_ptr	C++17	소유권 없음(no ownership)

2. shared_ptr

핵심 개념

- 소유권 공유의 개념을 구현한 스마트 포인터, 참조 계수 방식으로 객체의 수명을 관리.
- Raw Pointer의 2개의 크기를 가지게 됩니다.
- shared_ptr 생성시 참조계수등을 관리하는 제어 블록이 생성됩니다.
- 참조계수의 증가/감소는 원자적 연산으로 수행됩니다.

기본 코드

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

class Car
{
    int color;
public:
    Car()  { cout << "Car()" << endl; }
    ~Car() { cout << "~Car()" << endl; }
    void Go() { cout << "Car Go" << endl; }
};
int main()
{
    shared_ptr<Car> p1(new Car);
    p1->Go();
    shared_ptr<Car> p2 = p1;
    cout << p2.use_count() << endl;
    p1.reset();
    cout << p2.use_count() << endl;
    cout << p2       << endl;
    cout << p2.get() << endl;
}
```

▌ . 연산과 -> 연산

shared_ptr 사용시 . 연산을 사용하면 shared_ptr 자체의 멤버에 접근할 수 있습니다.

-> 연산을 사용하면 대상 객체의 멤버에 접근할 수 있습니다.

```
#include <iostream>
#include <memory>
#include "Car.h"
using namespace std;

int main()
{
    shared_ptr<Car> p1( new Car ); //1

    p1->Go();  // Car 의 멤버 접근.

    Car* p = p1.get();
    cout << p << endl;

    shared_ptr<Car> p2 = p1; // 2
    cout << p1.use_count() << endl; // 2

    //p1 = new Car; // error
    p1.reset( new Car); // ok
    p1.reset();
    p1.swap(p2);

}
```

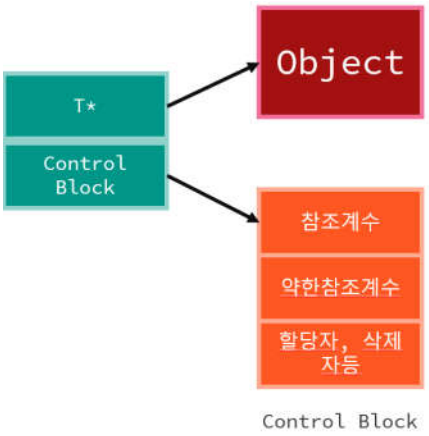
삭제자(delete) 변경하기

shared_ptr<>은 기본적으로 소멸자에서 delete를 사용해서 객체를 파괴 합니다. 함수, 함수객체, 람다표현식 등을 사용해서 객체의 파괴방식을 변경할 수 있습니다.

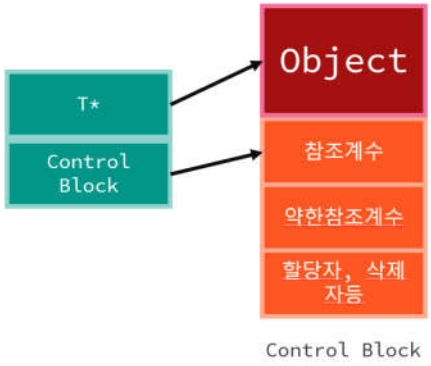
```
#include <iostream>
#include <memory>
using namespace std;

void del_arr(int* p)
{
    delete[] p;
}
int main()
{
    shared_ptr<int> p1(new int);                // delete
    shared_ptr<int> p2(new int[10], del_arr);
    shared_ptr<int> p3(new int[10], [](int* p) { delete[] p; });
}
```

make_shared

shared_ptr◇은 참조계수 기반으로 객체를 관리 하기 때문에 shared_ptr◇의 객체를 생성하면 참조계수 등을 관리하기 제어블럭이 같이 생성되게 됩니다. 이때, make_shared 를 사용하면 객체와 제어블럭을 묶어서 메모리를 할당 할 수 있기 때문에 보다 효율적으로 메모리를 사용할 수 있습니다.



make_shared_를 사용하지 않은 경우



make_shared_를 사용하는 경우

operator new() 함수를 재정의해서 테스트해 볼 수 있습니다.

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

class Car
{
    int color;
public:
    Car() { cout << "Car()" << endl; }
    ~Car() { cout << "~Car()" << endl; }
};
void* operator new(size_t sz)
{
    cout << "operator new : " << sz << endl;
    return malloc(sz);
}
int main()
```

```
{
    shared_ptr<Car> p1(new Car);    // 1. Car 객체 생성
                    // 2. control block 생성
    shared_ptr<Car> p2 = make_shared<Car>();// sizeof(Car) + sizeof(control block)을
한번에 생성
}
```

enable_shared_from_this

enable_shared_from_this 를 사용하면 객체 안에서 this를 사용해서 객체 자신의 참조계수를 증가할수 있습니다.

```
#include <iostream>
#include <memory>
#include <thread>
using namespace std;
class Worker : public enable_shared_from_this<Worker>
{
    int data = 0;
    shared_ptr<Worker> holdme;
public:
    ~Worker() { cout << "~Worker()" << endl;  }
    void run()
    {
        holdme = shared_from_this();
        thread t(&Worker::main, this);
        t.detach();
    }
    void main()
    {
        cout << "Worker main" << endl;
        this_thread::sleep_for(1s);
        data = 100;
        cout << "Worker end" << endl;
        holdme.reset();
    }
};
int main()
{
    {
        shared_ptr<Worker> p(new Worker);
        p->run();
    }
    cout << "main end" << endl;
    getchar();
}
```

항목 6-2

smart pointer #2

학습 목표

- 상호 참고와 weak_ptr
- unique_ptr

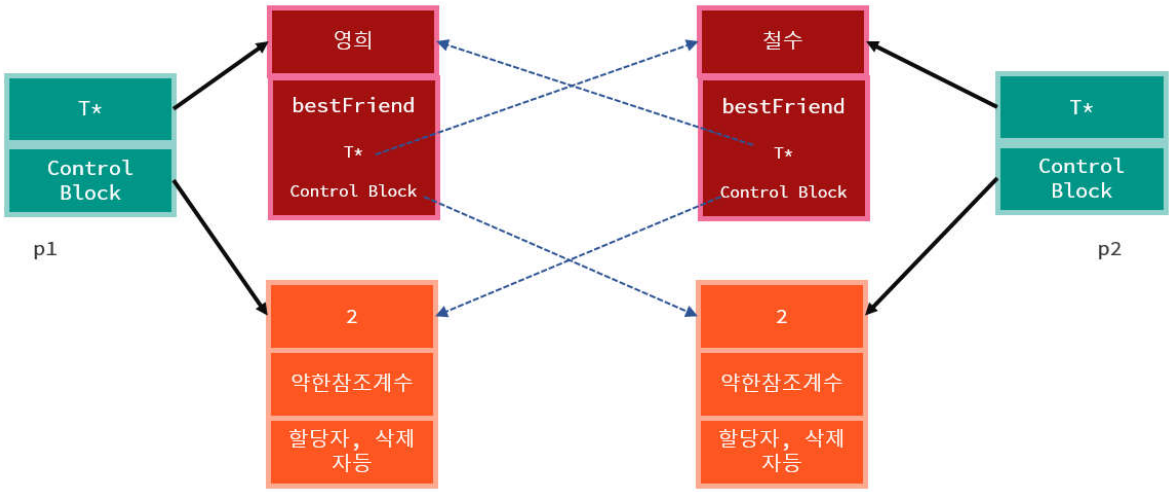
1. 스마트 포인터의 상호 참조 문제

▌ 핵심 개념

shared_ptr<>을 사용할 때 상호 참조가 발생하면 메모리 누수가 발생될 수 있습니다.

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

class People
{
public:
    People(string s) : name(s) {}
    ~People() { cout << name << " 파괴" << endl; }

    shared_ptr<People> bestFriend;
private:
    string name;
};

int main()
{
    shared_ptr<People> p1(new People("영희"));
    shared_ptr<People> p2(new People("철수"));

    p1->bestFriend = p2;
    p2->bestFriend = p1;
}
```

위 코드 실행시 메모리 그림이 다음과 같습니다.



이 문제를 해결하려면 참조 계수가 증가 하지 않은 스마트 포인터가 필요합니다. 다음 항목을 참고하세요

2. weak_ptr

▌ 핵심 개념

객체의 수명에 관련된 참조 계수를 증가하지 않은 스마트 포인터.

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

class Car
{
    int color;
public:
    Car() { cout << "Car()" << endl; }
    ~Car() { cout << "~Car()" << endl; }
};

int main()
{
    shared_ptr<Car> sp1 = make_shared<Car>();
    cout << sp1.use_count() << endl; // 1

    shared_ptr<Car> sp2 = sp1;
    cout << sp1.use_count() << endl; // 2

    weak_ptr<Car> wp1 = sp1;
    cout << sp1.use_count() << endl; // 2

    sp2.reset();
    cout << sp1.use_count() << endl; // 1
}
```

expired

weak_ptr<>의 expired() 멤버 함수를 사용하면 객체가 파괴 되었는지를 조사할 수 있습니다.

```
int main()
{
    shared_ptr<Car> sp1 = make_shared<Car>();
    weak_ptr<Car> wp1 = sp1;

    sp1.reset();

    if (wp1.expired())
        cout << "객체가 파괴되었습니다." << endl;
    else
    {
        cout << "객체가 파괴 되지 않았습니다." << endl;

        wp->Go(); // error, weak_ptr<>은 -> 연산자를 제공하지 않습니다.
    }
}
```

하지만, weak_ptr<>은 -> 연산자를 제공하지 않기 때문에 객체에 접근할 수는 없습니다. 따라서, weak_ptr을 사용해서 객체에 접근하려면 다시 shared_ptr<>을 만들어서 사용해야 합니다.

▌ weak_ptr◇로 객체에 접근 하는 방법.

weak_ptr◇로 객체를 가리킬 때 객체에 접근하려면 weak_ptr◇의 lock() 함수를 사용해서 shared_ptr◇의 객체를 생성해야 합니다.

```
int main()
{
    shared_ptr<Car> sp1 = make_shared<Car>();
    weak_ptr<Car> wp1 = sp1;

    shared_ptr<Car> sp2 = wp1.lock();
    if (sp2)
        sp2->Go();
}
```

▌ weak_ptr◇을 사용한 상호 참조 문제의 해결

weak_ptr◇를 사용하면 shared_ptr◇의 상호 문제를 해결 할 수 있습니다. 상호 참조가 발생하는 경우 shared_ptr◇대신 weak_ptr◇을 사용하면 됩니다.

```
class People
{
public:
    People(string s) : name(s) {}
    ~People() { cout << name << " 파괴" << endl; }

    void setBestFriend(weak_ptr<People> wp) { bestFriend = wp; }
    void print_bestFriend()
    {
        shared_ptr<People> sp = bestFriend.lock();

        if (sp)
            cout << sp->name << endl;
    }
private:
    string name;
    weak_ptr<People> bestFriend;
};
int main()
```

```
{
    shared_ptr<People> p1(new People("영희"));
    shared_ptr<People> p2(new People("철수"));

    p1->setBestFriend(p2);
    p2->setBestFriend(p1);
    p1->print_bestFriend();
}
```

3. unique_ptr

▌ 핵심 개념

자원에 대한 독점적 소유권을 가진 스마트 포인터. 복사는 불가능하지만, 이동은 가능합니다.

▌ 기본 코드

weak_ptr◇를 사용하면 shared_ptr◇의 상호 문제를 해결 할 수 있습니다. 상호 참조가 발생하는 경우 shared_ptr◇대신 weak_ptr◇을 사용하면 됩니다.

```
int main()
{
    unique_ptr<Car> up1 = make_unique<Car>();
    unique_ptr<Car> up2 = up1;      // error
    unique_ptr<Car> up3 = move(up1);// ok
}
```

▌ unique_ptr◇ 과 배열

배열 형식으로 메모리를 할당할 경우 unique_ptr◇의 타입인자는 배열형식으로 전달해야 합니다. 또한, 배열 타입의 unique_ptr◇은 [] 연산은 가능하지만 * 연산은 사용할 수 없습니다.

```
int main()
{
    unique_ptr<int> up1(new int);
    unique_ptr<int> up2(new int[10]);   // bug.
    unique_ptr<int[]> up3(new int[10]);     // ok.
                        // int 타입의 unique_ptr
    *up1 = 10;  // ok
    up1[0] = 10;    // error, [] 연산자는 배열타입의 unique_ptr만 가능합니다.
            // 배열 타입의 unique_ptr
    *up3 = 10;  // error. 배열타입의 unique_ptr의 경우 *을 사용할수 없습니다.
    up3[0] = 10;    // ok.
}
```

삭제자 변경

unique_ptr의 경우, 삭제자를 변경하려면 삭제자의 타입을 템플릿 인자로 전달하면 됩니다.

```
struct Freer
{
    void operator()(void* p) { free(p); }
};

int main()
{
    unique_ptr<int, Freer> up1(new malloc(100));

    // unique_ptr의 삭제자로 람다 표현식를 사용한 경우
    auto freer = [](void* p) { free(p); };
    unique_ptr<int, decltype(freer)> up2(new malloc(100), freer);
}
```

항목 7-1

thread

학습 목표

- thread
- mutex & lock_guard
- thread_local
- promise & future
- packaged_task
- async

1. thread

▌ thread 생성

C++에서 스레드를 생성하려면 thread 객체를 생성하면 됩니다 또한, 스레드를 생성한 후에는 join() 하거나 detach() 해야 합니다.

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;
using namespace std::chrono;

// Step1. thread

int foo(int n)
{
    cout << "start foo, thread id : " << this_thread::get_id() << endl;
    this_thread::sleep_for(2s);
    cout << "end foo" << endl;
    return 100;
}
int main()
{
    thread t(foo, 10);

    cout << "main id : " << this_thread::get_id() << endl;
    t.join();
}
```

▌ thread 와 callable object

다양한 종류의 callable object를 스레드 함수로 사용할 수 있습니다.

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;
using namespace std::chrono;
```

```
void f1() {}

struct FOO
{
    void operator()(int a) {}
};
class Test
{
public:
    void foo() {}
};
int main()
{
    thread t1(f1);          // 일반 함수
    thread t2(FOO(), 10);   // 함수 객체

    Test test;
    thread t3(&Test::foo, &test);// 멤버 함수
    thread t4([](int a) {}, 10); // 람다 표현식

    t1.detach();
    t2.detach();
    t3.detach();
    t4.detach();
}
```

▌ 동기화 객체와 lock_guard

뮤텍스, 세마 포어, 조건변수 등 대부분의 동기화 요소 역시 제공됩니다. 또한, 뮤텍스 등을 사용할때는 lock_guard 등을 사용하는 것이 좋습니다.

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

mutex m;

void foo()
{
```

```
    lock_guard<mutex> g(m);

    // ......
}
int main()
{
    thread t1(foo);
    thread t2(foo);
    t1.join();
    t2.join();
}
```

▌ thread local storage (thread specific storage)

스레드별 메모리 공간을 할당 하려면 thread_local 키워드를 사용합니다.

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

int next3times()
{
    thread_local static int n = 0;
    n = n + 3;
    return n;
}
void foo(string name)
{
    cout << name << " : " << next3times() << endl;
    cout << name << " : " << next3times() << endl;
    cout << name << " : " << next3times() << endl;
}

int main()
{
    thread t1(foo, "A"s);
    thread t2(foo, "B"s);

    t1.join();
    t2.join();
```

```
}
```

▌ promise & future

주 스레드에서 새로운 스레드의 결과 값을 꺼내 올 때는 promise 객체와 future 객체를 활용합니다.

```
#include <iostream>
#include <future>
#include <chrono>
using namespace std;
using namespace std::chrono;

int foo(promise<int>& p)
{
    cout << "start foo, thread id : " << this_thread::get_id() << endl;
    this_thread::sleep_for(2s);
    cout << "end foo" << endl;
    p.set_value(100);
    return 100;
}

int main()
{
    promise<int> p;
    thread t(foo, ref(p));

    future<int> ft = p.get_future();
    cout << "main id : " << this_thread::get_id() << endl;

    int n = ft.get();
    cout << "value : " << n << endl;
    t.join();
}
```

▌ packaged_task

packaged_task 객체를 사용하면 thread 의 리턴 값을 promise 객체를 사용해서 전달할 수 있습니다.

```
#include <iostream>
#include <future>
#include <chrono>
#include <string>
using namespace std;
using namespace std::chrono;

string foo()
{
    cout << "start packaged test : " << this_thread::get_id() << endl;
    this_thread::sleep_for(2s);
    cout << "end packaged test" << endl;

    return "hello"s;    // p.set_value("hello"s)
}

int main()
{
    packaged_task<string()> p_task(foo); // promise를 포함

    auto future = p_task.get_future();
    cout << "main : " << this_thread::get_id() << endl;

    thread t1(move(p_task));
    auto ret = future.get();
    cout << "result : " << ret << endl;

    t1.join();
}
```

▌ async

async() 함수를 사용하면 스레드를 쉽게 생성하고 결과값을 가져올 수 있습니다.

```
#include <iostream>
#include <string>
#include <future>
#include <chrono>
using namespace std;
using namespace std::chrono;

string foo()
```

```
{
    cout << "start foo : " << this_thread::get_id() << endl;
    this_thread::sleep_for(2s);
    cout << "end foo" << endl;

    return "hello"s;
}
int main()
{
    cout << "main thread : " << this_thread::get_id() << endl;

    // 1. 리턴값을 받지 않은 경우
    //async(foo);

    // 2. 리턴값을 받는 경우
    future<string> ft = async(foo);

    // 3. launch option 사용
    //future<string> ft = async(launch::deferred,  foo);

    cout << "main end" << endl;

    auto r = ft.get();
    cout << "value : " << r << endl;
}
```