# P1.S13

| | |
|---|---|
| 작성일시 | @2024년 11월 11일 오전 5:00 |
| 강의 번호 | C++ 언리얼 |
| 유형 | 강의 |
| 복습 | ☐ |
| 날짜 | @2024년 11월 11일 |

auto

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
#include <algorithm>

// 오늘의 주제 : auto

class Knight
{
public:
    int _hp;
};

template<typename T>
void Print(T t)
{
    cout << t << endl;
}

int main()
{
```

```
// Modern C++ (C++11)

/*int a = 3;
float b = 3.14f;
double c = 1.23;
Knight d = Knight();
const char* e = "rookiss";*/


auto a = 3;
auto b = 3.14f;
auto c = 1.23;
auto d = Knight();
auto e = "rookiss";

Print(3);
Print("rookiss");

// auto는 일종의 조커카드
// 형식 연역 (type deduction)
// -> 말이되게 잘 맞춰봐~ (추론)
// 추론 규칙은 생각보다 복잡해질 수 있다

// 주의!
// 기본 auto는 const, & 무시!!!!!!
int& reference = a;
const int cst = a;

auto test1 = reference;
auto test2 = cst;

// 실수 예시)
vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);

for (vector<int>::size_type i = 0; i < v.size(); i++)
```

```
    {
        auto& data = v[i]; // &를 붙여서 참조를 유지해야한다고 힌트를
        data = 100;
    }

    // 아무튼 이제 기존의 타입은 잊어버리고 auto만 사용할래~
    // NO! (주관적 생각임)
    // -> 타이핑이 길어지는 경우 OK
    // -> 가독성을 위해 일반적으로는 놔두는게 좋다

    map<int, int> m;
    auto ok = m.insert(make_pair(1, 100));

    for (auto it = v.begin(); it != v.end(); ++it)
    {
        cout << *it << endl;
    }

    return 0;
}
```

중괄호 초기화 { }

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
#include <algorithm>

// 오늘의 주제 : 중괄호 초기화 { }

class Knight
{
public:
    Knight()
```

```
    {

    }

    Knight(int a, int b)
    {
        cout << "Knight(int a, int b)" << endl;
    }

    Knight(initializer_list<int> li)
    {
        cout << "Knight(initializer_list)" << endl;
    }
};

int main()
{
    // 중괄호 초기화 { }
    int a = 0;
    int b(0);
    int c{ 0 };

    Knight k1;
    Knight k2 = k1; // 복사 생성자 (대입 연산자X)

    // 복사 생성자랑 대입 연산자는 잘 구분하자
    //Knight k3; // 기본 생성자
    //k3 = k1; // 대입 연산자

    Knight k3{ k2 };

    vector<int> v1;
    v1.push_back(1);
    v1.push_back(2);
    v1.push_back(3);

    vector<int> v2(10, 1); // (개수, 값)
```

```
    // 중괄호 초기화

    // [장점]
    // 1) vector 등 container와 잘 어울린다
    vector<int> v3{ 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5};

    // 2) 축소 변환 방지
    //int x = 0;
    //double y{ x };

    // 3) Bonus
    // k4라는 나이트타입 변수를 만들고 기본생성자를 호출할라했는데,
    // 숙련도 부족으로 함수를 선언한게 되어버림
    //Knight k4();
    Knight k4{};


    // [단점이라 할 수 있는 부분]
    // 벡터와 같이 무제한으로 데이터를 받아주고싶으면 initializer_list
    Knight k5{ 1, 2, 3, 4, 5 };

    // 인자 두개 받는 생성자를 만들어줬음에도 불구하고 initializer_list
    // 따라서 initializer_list가 생태계 파괴를 하는 느낌
    Knight k6{ 1, 2 };


    // 괄호 초기화 ()를 기본으로 간다
    // - 전통적인 C++ (거부감이 없음)
    // - vector 등 특이한 케이스에 대해서만 { } 사용

    // 중괄호 초기화 { }를 기본으로 간다
    // - 초기화 문법의 일치화
    // - 축소 변환 방지

    return 0;
}
```

nullptr

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
#include <algorithm>

// 오늘의 주제 : nullptr

class Knight
{
public:
    void Test()
    {

    }
};

Knight* FindKnight()
{
    // TODO

    return nullptr;
}

void Test(int a)
{
    cout << "Test(int)" << endl;
}

void Test(void* ptr)
{
    cout << "Test(*)" << endl;
}
```

```
// NullPtr 구현
class NullPtr
{
public:
    // 그 어떤 타입의 포인터와도 치환 가능
    template<typename T>
    operator T* () const
    {
        return 0;
    }

    // 그 어떤 타입의 멤버 포인터와도 치환 가능
    template<typename C, typename T>
    operator T C::* () const
    {
        return 0;
    }

private:
    void operator&() const; // 주소값 &을 막는다
} _nullPtr; // 선언하자마자 객체 만들기

int main()
{
    int* ptr = NULL; // NULL은 그냥 0을 #define해놓은거임

    // 1) 오동작
    // 아래 둘다 Test(int a)를 호출함
    Test(0);
    Test(NULL);

    Test(nullptr); // 없는 포인터를 사용하고싶으면 무조건 nullptr

    // 2) 가독성
    auto knight = FindKnight();
    if (knight == nullptr) // nullptr 덕분에 포인터인지 힌트
```

```
    {

    }

    //nullptr_t whoami = nullptr;

    void (Knight:: * memFunc)();
    memFunc = &Knight::Test;

    if (memFunc == _nullPtr)
    {

    }

    return 0;
}
```

using

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
#include <algorithm>

// 오늘의 주제 : using

typedef vector<int>::iterator VecIt;

typedef __int64 id;

// 1) 직관성
typedef void (*MyFunc)();
using MyFunc3 = void(*)();
```

```
// 2) 템플릿

template<typename T>
using List = std::list<T>;

// using에 비해서 장점이 하나도 없음
template<typename T>
struct List2
{
    typedef std::list<T> type;
};

int main()
{
    id playerId = 0;

    List<int> li;
    li.push_back(1);
    li.push_back(2);
    li.push_back(3);


    List2<int>::type li2;

    return 0;
}
```

enum class

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
#include <algorithm>
```

```
// 오늘의 주제 : enum class

enum PlayerType
{
    PT_None,
    PT_KNIGHT,
    PT_ARCHER,
    PT_MAGE
};

enum MonsterType
{
    MT_None,
    MT_KNIGHT,
};

enum class ObjectType
{
    Player,
    Monster,
    Projectile
};

enum class ObjectType2
{
    Player,
    Monster,
    Projectile
};

int main()
{
    // enum class (scoped enum)
    // 1) 이름공간 관리 (scoped)
    // 2) 암묵적인 변환 금지

    // 암묵적 변환이 안돼서 어거지로 해줘야함
```

```
    double value = static_cast<double>(ObjectType::Monster);

    int choice;
    cin >> choice;

    if (choice == static_cast<int>(ObjectType::Monster))
    {

    }

    unsigned int bitFlag;
    bitFlag = (1 << static_cast<int>(ObjectType::Player));

    // 따라서 enum이나 enum class나 장단점이 있음
    // 둘다 사용할 줄 알아야함

    return 0;
}
```

delete

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
#include <algorithm>

// 오늘의 주제 : delete (삭제된 함수)

class Knight
{
public:
    // delete를 사용하는 새로운 방법
    void operator=(const Knight& k) = delete;
```

```
private:
    // 정의되지 않은 비공개(private) 함수
    // C++ 11에서 어떤 함수를 사용하지 못하게 막는 방법
    // private으로 하고, 선언만 해줌(구현 안함)
    // 옛날 방법
    //void operator=(const Knight& k);


    // 권한 열어주는 friend
    // 모든 것을 뚫는 창 vs 절대 방패
    //friend class Admin;

private:
    int _hp = 100;
};

class Admin
{
public:
    void CopyKnight(const Knight& k)
    {
        Knight k1;
        // 복사 연산
        k1 = k;
    }
};


int main()
{
    Knight k1;

    Knight k2;


    // 복사 연산자
    //k1 = k2;
```

```
    return 0;
}
```

override, final

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
#include <algorithm>

// 오늘의 주제 : override, final

class Creature
{
public:
    virtual void Attack()
    {
        cout << "Creature!" << endl;
    }
};

class Player : public Creature
{
public:
    virtual void Attack() override
    {
        cout << "Player!" << endl;
    }
};

class Knight : public Player
{
public:
```

```
    // 재정의(override)
    virtual void Attack() override
    {
        cout << "Knight!" << endl;
    }

private:
    int _stamina = 100;

};

int main()
{
    // virtual만 붙여주던 기본문법
    // knight가 최초로 만든 것인지 그 부모가 만든걸 재정의 한 것인지 알기
    Player* p = new Knight();

    p->Attack();

    // override 하고 있을때는 함수 이름 옆에 override를 붙여주자!

    // final을 붙여주면 상속받은애들이 override를 못하게 된다

    return 0;
}
```

오른값 참조(rvalue reference)

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
#include <algorithm>

// 오늘의 주제 : 오른값(rvalue) 참조와 std::move
```

```
class Pet
{

};

class Knight
{
public:

    void PrintInfo() const
    {
        // read only 목적
    }

public:
    Knight()
    {
        cout << "Knight()" << endl;
    }

    // 복사 생성자
    Knight(const Knight& knight)
    {
        cout << "const Knight&" << endl;
    }

    ~Knight()
    {
        if (_pet)
            delete _pet;
    }

    // 이동 생성자
    Knight(Knight&& knight)
    {

    }
```

```
    // 복사 대입 연산자
    void operator=(const Knight& knight)
    {
        cout << "operator=(const Knight&)" << endl;

        _hp = knight._hp;
        //_pet = knight._pet; // 얕은 복사

        if(knight._pet) // 깊은 복사
            _pet = new Pet(*knight._pet);
    }

    // 이동 대입 연산자
    void operator=(Knight&& knight) noexcept // 경고 밑줄을 없애
    {
        cout << "operator=(Knight&&)" << endl;

        // 얕은 복사
        _hp = knight._hp;
        _pet = knight._pet;

        knight._pet = nullptr;
    }


public:
    int _hp = 100;
    Pet* _pet = nullptr;
};

void TestKnight_Copy(Knight knight) {}
void TestKnight_LValueRef(Knight& knight) {}
void TestKnight_ConstLValueRef(const Knight& knight) { knight
void TestKnight_RValueRef(Knight&& knight) {} // 이동 대상!


int main()
```

```
{
    // 왼값(lvalue) vs 오른값(rvalue)
    // - lvalue : 단일식을 넘어서 계속 지속되는 개체
    // - rvalue : lvalue가 아닌 나머지 (임시 값, 열거형, 람다, i++ 

    int a = 3;

    Knight k1;

    TestKnight_Copy(k1);
    TestKnight_LValueRef(k1);
    //TestKnight_LValueRef(Knight()); // 한번사용하고 없어질 애라서

    TestKnight_ConstLValueRef(Knight());

    TestKnight_RValueRef(Knight());

    // 강제로 k1의 원본을 건내줘보자
    TestKnight_RValueRef(static_cast<Knight&&>(k1));


    Knight k2;
    k2._pet = new Pet();
    k2._hp = 1000;

    // 원본은 날려도 된다 <<는 힌트를 주는 쪽에 가깝다!
    // 그냥 복사되는건 생각보다 무거울 수 있는데
    // 이동되는건 정보만 슉 빼오는거라 빠르다
    Knight k3;
    //k3 = static_cast<Knight&&>(k2);

    k3 = std::move(k2); // 오른값 참조로 캐스팅
    // std::move의 본래 이름 후보 중 하나가 rvalue_cast


    // 세상에 하나만 존재하는 포인터
    std::unique_ptr<Knight> uptr = std::make_unique<Knight>()
    // 하나만 존재하는걸 이동시켜주는 경우 오른값 참조로 이동시켜주면 좋다
```

```
    std::unique_ptr<Knight> uptr2 = std::move(uptr);


    return 0;
}
```

전달 참조 (forwarding reference)

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
#include <algorithm>

// 오늘의 주제 : 전달 참조 (forwarding reference)

class Knight
{
public:
    Knight() { cout << "기본 생성자" << endl; }
    Knight(const Knight&) { cout << "복사 생성자" << endl; }
    Knight(Knight&&) noexcept { cout << "이동 생성자" << endl; }
};

void Test_RValueRef(Knight&& k) // 오른값 참조
{

}

void Test_Copy(Knight k)
{

}

// 왼값을 넣어주면 왼값 참조해주고, 오른값을 넣어주면 오른 참조를 해주는 전
```

```
template<typename T>
void Test_ForwardingRef(T&& param) // 전달 참조
{
    // 넘겨준 값이 왼값 참조라면 복사
    // 오른값 참조라면 이동
    Test_Copy(std::forward<T>(param));
}

int main()
{
    // 보편 참조(universal reference)
    // 전달 참조(forwarding reference) C++17

    // &&   &를 두 번 -> 오른값 참조 말고 또 있다

    Knight k1;

    //Test_RValueRef(std::move(k1));

    //Test_ForwardingRef(k1); // 왼값 참조로 받아줌
    //Test_ForwardingRef(std::move(k1)); // 오른값 참조로 받아줌

    auto&& k2 = k1;
    auto&& k3 = std::move(k1);

    // 공통점 : 형식 연역 (type deduction)이 일어날 때


    // 전달 참조를 구별하는 방법
    // ----------------------------------------

    Knight& k4 = k1; // 왼값 참조
    Knight&& k5 = std::move(k1); // 오른값 참조

    // 오른값 : 왼값이 아니다 = 단일식에서 벗어나면 사용 X
    // 오른값 참조 : 오른값만 참조할 수 있는 참조 타입
    //Test_RValueRef(k5); // 에러
    //Test_RValueRef(std::move(k5));
```

```
    // Test_ForwardingRef의 param으로 오른값 k1을 넣어줘도 함수 안에
    // 따라서 std::move 로 오른값으로 만들어주지 않으면
    // 이동생성자 말고 복사생성자가 호출되고 있다는걸 알 수 있음
    // 반면에 왼값을 넣어서 작업해주고싶은게 생길 수 있다
    // 그렇다면 두 가지 방향을 따로 설정해줄 필요가 생기는데 이럴때 std::
    Test_ForwardingRef(k1);
    Test_ForwardingRef(std::move(k1));

    return 0;
}
```

람다 (lambda)

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
#include <algorithm>

// 오늘의 주제 : 람다(lambda)

// 함수 객체를 빠르게 만드는 문법

enum class ItemType
{
    None,
    Armor,
    Weapon,
    Jewelry,
    Consumable
};

enum class Rarity
```

```
{
    Common,
    Rare,
    Unique
};

class Item
{
public:
    Item() {}
    Item(int itemId, Rarity rarity, ItemType type)
        : _itemId(itemId), _rarity(rarity), _type(type)
    {

    }

public:
    int _itemId = 0;
    Rarity _rarity = Rarity::Common;
    ItemType _type = ItemType::None;
};

int main()
{
    vector<Item> v;
    v.push_back(Item(1, Rarity::Common, ItemType::Weapon));
    v.push_back(Item(2, Rarity::Common, ItemType::Armor));
    v.push_back(Item(3, Rarity::Rare, ItemType::Jewelry));
    v.push_back(Item(4, Rarity::Unique, ItemType::Weapon));

    // 람다 = 함수 객체를 손쉽게 만드는 문법
    // 람다 자체로 C++11 에 '새로운' 기능이 들어간 것은 아니다
    {
        struct IsUniqueItem
        {
            bool operator()(Item& item)
            {
                return item._rarity == Rarity::Unique;
```

```
        }
    };

    // 클로저 (closure) = 람다에 의해 만들어진 실행시점 객체
    auto isUniqueLambda = [](Item& item)
        {
            return item._rarity == Rarity::Unique;
        }; // 람다 표현식(lambda expression)

    /*auto findIt = std::find_if(v.begin(), v.end(), IsUn
    if (findIt != v.end())
        cout << "아이템ID: " << findIt->_itemId << endl;*/

        /*auto findIt = std::find_if(v.begin(), v.end(),
        if (findIt != v.end())
            cout << "아이템ID: " << findIt->_itemId << endl

            // 아래와 같이 람다로 한방에 만들 수 있다
            /*auto findIt = std::find_if(v.begin(), v.end
                [](Item& item){ return item._rarity == Ra
            if (findIt != v.end())
                cout << "아이템ID: " << findIt->_itemId <<


    struct FindItemByItemId
    {
        FindItemByItemId(int itemId) : _itemId(itemId)
        {

        }

        bool operator()(Item& item)
        {
            return item._itemId == _itemId;
        }

        int _itemId;
    };
```

```
int itemId = 4;

// [] = 캡쳐(capture) : 함수 객체 내부에 변수를 저장하는 개념
// 사진을 찰칵 [캡쳐]하듯... 일종의 스냅샷을 찍는다고 이해
// 기본 캡처 모드 : 값(복사) 방식(=) 참조 방식(&)
// 변수마다 캡처 모드를 지정해서 사용 가능 : 값 방식[name], 참
// =, & 하나로 퉁치는건 최대한 지양하자

auto findByItemIdLambda = [=](Item& item)
    {
        return item._itemId == itemId;
    };

// 참조방식 (&) 을 사용하면 데이터의 주소로 접근하기때문에
// 후에 itemId = 10; 이런식으로 변경이 있으면
// 변경한 값이 영향을 주게 된다

//auto findIt = std::find_if(v.begin(), v.end(), Find
/*auto findIt = std::find_if(v.begin(), v.end(), findI

if (findIt != v.end())
    cout << "아이템ID: " << findIt->_itemId << endl;*/


struct FindItem
{
    FindItem(int itemId, Rarity rarity, ItemType type
        : _itemId(itemId), _rarity(rarity), _type(typ
    {

    }

    bool operator()(Item& item)
    {
        return item._itemId == _itemId && item._rarit
    }
```

```
            int _itemId;
            Rarity _rarity;
            ItemType _type;
        };

        //int itemId = 4;
        Rarity rarity = Rarity::Unique;
        ItemType type = ItemType::Weapon;

        auto findItemLambda = [&itemId, &rarity, &type](Item&
            {
                return item._itemId == itemId && item._rarity
            };

        auto findIt = std::find_if(v.begin(), v.end(), FindIt

        if (findIt != v.end())
            cout << "아이템ID: " << findIt->_itemId << endl;
    }



    {
        class Knight
        {
        public:
            auto ResetHpJob()
            {
                auto f = [this]()
                    {
                        _hp = 200;
                    };

                return f;
            }

        public:
            int _hp = 100;
        };
```

```
        Knight* k = new Knight();
        auto job = k->ResetHpJob();
        delete k;
        job();
        // 크래시가 안나는데 진짜 너무 끔찍한 상황
        // this 를 보고 주의하게 만들어주는 수밖에 없고 조심하자
    }

    return 0;
}
```

스마트 포인터

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
#include <algorithm>

// 오늘의 주제 : 스마트 포인터 (smart pointer)

class Knight
{
public:
    Knight() { cout << "Knight 생성" << endl; }
    ~Knight() { cout << "Knight 소멸" << endl; }

    /*void Attack()
    {
        if (_target)
        {
            _target->_hp -= _damage;
            cout << "HP: " << _target->_hp << endl;
        }
```

```
    }*/

    void Attack()
    {
        if (_target.expired() == false)
        {
            shared_ptr<Knight>sptr = _target.lock();
            sptr->_hp -= _damage;
            cout << "HP: " << sptr->_hp << endl;
        }
    }

public:
    int _hp = 100;
    int _damage = 10;
    //shared_ptr<Knight> _target = nullptr;
    weak_ptr<Knight> _target; // 객체가 날라갔는지 안날라갔는지 확인
};

class RefCountBlock
{
public:
    int _refCount = 1;

    // weak_ptr를 쓰면 _weakCount가 추가된다
    int _weakCount = 1;
};

template<typename T>
class SharedPtr
{
public:
    SharedPtr() {}

    SharedPtr(T* ptr) : _ptr(ptr)
    {
        if (_ptr != nullptr)
        {
```

```
            _block = new RefCountBlock();
            cout << "RefCount : " << _block->_refCount << end
        }
    }

    // 복사 생성자
    SharedPtr(const SharedPtr& sptr) : _ptr(sptr._ptr), _bloc
    {
        if (_ptr != nullptr)
        {
            _block->_refCount++;
            cout << "RefCount : " << _block->_refCount << end
        }
    }

    // 복사 대입 연산자
    void operator=(const SharedPtr& sptr)
    {
        _ptr = sptr._ptr;
        _block = sptr._block;

        if (_ptr != nullptr)
        {
            _block->_refCount++;
            cout << "RefCount : " << _block->_refCount << end
        }
    }

    ~SharedPtr()
    {
        if (_ptr != nullptr)
        {
            _block->_refCount--;
            cout << "RefCount : " << _block->_refCount << end

            if (_block->_refCount == 0)
            {
                delete _ptr;
```

```
                delete _block;
                cout << "Delete Data" << endl;
            }
        }
    }

public:
    T* _ptr;
    RefCountBlock* _block = nullptr;
};

int main()
{
    //Knight* k1 = new Knight();
    //Knight* k2 = new Knight();

    //k1->_target = k2;

    //delete k2;

    //// 끔찍한 일이 생길것이다
    //k1->Attack();


    // 스마트 포인터 : 포인터를 알맞는 정책에 따라 관리하는 객체 (포인터를
    // shared_ptr, weak_ptr, unique_ptr


    // [SharedPtr 만들어보기]
    // 복사 생성자
    //SharedPtr<Knight> k1(new Knight());
    //SharedPtr<Knight> k2 = k1;

    // 복사 대입
    /*SharedPtr<Knight> k2;
    {
        SharedPtr<Knight> k1(new Knight());
        k2 = k1;
```

```
}*/


// [shared_ptr 써보기]
// make_shared는 Knight를 만들어줌과 동시에 같은 메모리에 위치시켜
/*shared_ptr<Knight> k1 = make_shared<Knight>();

{
    shared_ptr<Knight> k2 = make_shared<Knight>();
    k1->_target = k2;
}

k1->Attack();*/


// k1 [   2]
// k2 [   1] 순환구조라 1 이하로 떨어지지 않는다
// 즉 소멸 되지를 않는다
//shared_ptr<Knight> k1 = make_shared<Knight>();

//shared_ptr<Knight> k2 = make_shared<Knight>();
//k1->_target = k2;
//k2->_target = k1;

//k1->Attack();

//// 이렇게 억지로 끊어줘야함
//k1->_target = nullptr;
//k2->_target = nullptr;


// [weak_ptr] 써보기
// k2가 소멸되고 타겟을 잃은 k1의 어택이 실행되지 않음
shared_ptr<Knight> k1 = make_shared<Knight>();

{
    shared_ptr<Knight> k2 = make_shared<Knight>();
    k1->_target = k2;
```

```
        k2->_target = k1;
    }

    k1->Attack();


    // [unique_ptr]
    // 하나밖에 없는 ptr
    unique_ptr<Knight> uptr = make_unique<Knight>();
    unique_ptr<Knight> uptr2 = uptr; // 이러면 에러
    unique_ptr<Knight> uptr2 = std::move(uptr); // 이렇게 이동시

    return 0;
}
```