P1.S7

작성일시	@2024년 9월 6일 오후 7:29
강의 번호	C++ 언리얼
유형	강의
복습	☐
날짜	@2024년 9월 6일

객체지향의 시작

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

// 오늘의 주제 : 객체지향의 시작

// 절차(procedural)지향 프로그래밍
// - procedure = 함수
// main
// - EnterLobby(PlayerInfo)
// -- CreatePlayer
// -- EnterGame(MonsterInfo)
// --- CreateMonsters
// --- EnterBattle

// 데이터 + 가공(로직, 동작)

// 객체지향 = 객체
// 객체란? 플레이어, 몬스터, GameRoom

// Knight를 설계해보자
// - 속성(데이터) : hp, attack, y, x
// - 기능(동작) : Move, Attack, Die

// class는 일종의 설계도
class Knight {
```

```
public:
    // 멤버 함수 선언
    void Move(int y, int x);
    void Attack();
    void Die() {
        hp = 0;
        cout << "Die" << endl;
    }

public:
    // 멤버 변수
    int hp;
    int attack;
    int posY;
    int posX;
};

void Move(Knight* knight, int y, int x) {
    knight->posY = y;
    knight->posX = x;
}

void Knight::Move(int y, int x) {
    posY = y;
    posX = x;
    cout << "Move" << endl;
}

void Knight::Attack() {
    cout << "Attack : " << attack << endl;
}

// Instantiate 객체를 만든다!
int main() {
    Knight k1;
    k1.hp = 100;
    k1.attack = 10;
    k1.posY = 0;
```

```
    k1.posX = 0;

    Knight k2;
    k2.hp = 80;
    k2.attack = 5;
    k2.posY = 1;
    k2.posX = 1;

    // Move(&k1, 2, 2);

    k1.Move(2, 2);
    k1.Attack();
    k1.Die();

    return 0;
}
```

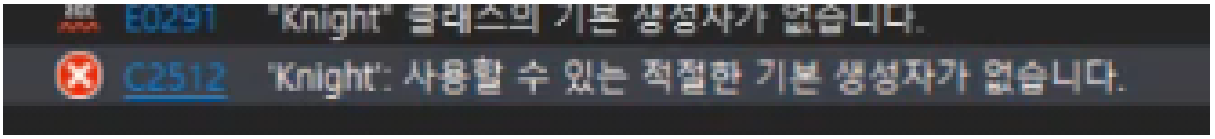
생성자와 소멸자

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

// 오늘의 주제 : 생성자와 소멸자

// [생성자(Constructor)와 소멸자(Destructor)]
// 클래스에 '소속'된 함수들을 멤버 함수라고 함
// 이 중에서 굉장히 특별한 함수 2종이 있는데, 바로 [시작]과 [끝]을 알리는
// - 시작(탄생) -> 생성자 (여러개 존재 가능)
// - 끝(소멸) -> 소멸자 (오직 1개만)

// [암시적(Implicit) 생성자]
// 생성자를 명시적으로 만들지 않으면
// 아무 인자도 받지 않는 [기본 생성자]가 컴파일러에 의해 자동으로 만들어짐
// -> 그러나 우리가 명시적(Explicit)으로 아무 생성자 하나를 만들면
// 자동으로 만들어지던 [기본 생성자]는 더이상 만들어지지 않음!

// class는 일종의 설계도
class Knight {
```

```
public:
    // [1] 기본 생성자 (인자가 없음)
    Knight() {
        cout << "Knight() 기본 생성자 호출" << endl;

        _hp = 100;
        _attack = 10;
        _posY = 0;
        _posX = 0;
    }

    // [2] 복사 생성자(자기 자신의 클래스 참조 타입을 인자로 받음)
    // (일반적으로 '똑같은' 데이터를 지닌 객체가 생성되길 기대한다
    // 직접 만들지 않아도 컴파일러가 자동으로 만들어주긴함. 하지만 참조나
    Knight(const Knight& knight) {
        _hp = knight._hp;
        _attack = knight._attack;
        _posX = knight._posX;
        _posY = knight._posY;
    }

    // [3] 기타 생성자

    // 이 중에서 인자를 1개만 받는 [기타 생성자]를
    // [타입 변환 생성자] 라고 부르기도 함

    // 명시적인 용도로만 사용할 것!! explicit을 앞에 붙임
    explicit Knight(int hp) {
        cout << "Knight(int) 생성자 호출" << endl;

        _hp = hp;
        _attack = 10;
        _posX = 0;
        _posY = 0;
    }

    Knight(int hp, int attack, int posX, int poxY) {
        cout << "Knight(int) 생성자 호출" << endl;
```

```
        _hp = hp;
        _attack = attack;
        _posX = posX;
        _posY = poxY;
    }

    // 소멸자(물결이 있으면 소멸자)
    ~Knight() {
        cout << "Knight() 기본 소멸자 호출" << endl;
    }

    // 멤버 함수 선언
    void Move(int y, int x);
    void Attack();
    void Die() {

        // 디스어셈블리 까보면 아래 두개는 같은 원리로 작동
        _hp = 0;
        this->_hp = 1;

        cout << "Die" << endl;
    }

public:
    // 멤버 변수
    int _hp;
    int _attack;
    int _posY;
    int _posX;
};

void Knight::Move(int y, int x) {
    _posY = y;
    _posX = x;
    cout << "Move" << endl;
}
```

```
void Knight::Attack() {
    cout << "Attack : " << _attack << endl;
}

void HelloKnight(Knight k) {
    cout << "Hello Knight" << endl;
}

// Instantiate 객체를 만든다!
int main() {
    Knight k1(100, 10, 0, 0);

    Knight k2(k1);

    // 생성을 함과 동시에 복사
    Knight k3 = k1;

    // 기본 생성자로 만든다음에 k1을 k4로 복사
    Knight k4;
    k4 = k1;

    k1.Move(2, 2);
    k1.Attack();
    k1.Die();

    // 암시적 형변환 -> 컴파일러가 알아서 바꿔치기
    int num = 1;

    float f = (float)num; // 명시적 = 우리가 코드로 num을 float바꾸
    double d = num; // 암시적 = 별말 안했는데 컴파일러가 알아서 처리

    Knight k5;
    k5 = (Knight)1;

    HelloKnight((Knight)5);

    return 0;
}
```



이런 경우는 명시적 생성자를 만들었기 때문에 기본생성자가 생성되지 않았고, 사용자가 기본생성자를 필요로 하는 생성을 한거

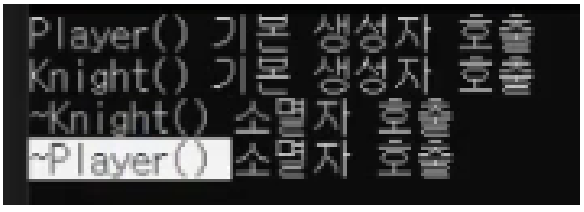
상속성

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

// 오늘의 주제 : 상속성

// 객체지향 (OOP Object Oriented Programming)
// - 상속성
// - 은닉성
// - 다형성

struct StatInfo {
    int hp;
    int attack;
    int defence;
};

// 상속(Inheritance) ? 부모 -> 자식에게 유산을 물려주는것

// 생성자(N)/소멸자(1)

// 생성자는 탄생을 기념해서 호출되는 함수?
// - Knight를 생성하면 -> Player의 생성자? Knight의 생성자?
// -> 솔로몬의 선택! 그냥 둘다 소환하자

// GameObject
// - Creature
// -- Player, Monster, Npc, Pet
// - Projectile
// -- Arrow, Fireball
```

```
// - Env

// Item
// - Weapon
// -- Sword, Bow
// - Armor
// -- Helmet, Boots, Armor
// - Consumable
// -- Potion, Scroll

class Player {
public:
    Player() {
        _hp = 0;
        _attack = 0;
        _defence = 0;
        cout << "Player() 기본 생성자 호출" << endl;
    }
    Player(int hp) {
        _hp = hp;
        _attack = 0;
        _defence = 0;
        cout << "Player(int hp) 생성자 호출" << endl;
    }
    ~Player() {
        cout << "Player() 소멸자 호출" << endl;
    }

    void Move() { cout << "Player Move 호출" << endl; }
    void Attack() { cout << "Player Attack 호출" << endl; }
    void Die() { cout << "Player Die 호출" << endl; }

public:
    int _hp;
    int _attack;
    int _defence;
};
```

```
class Knight : public Player{
public:
    Knight() {
        /*
        먼저 처리되는 영역이 존재함
        선처리 영역이라고도 함
        - 여기서 Player() 생성자를 호출
        */


        _stamina = 100;
        cout << "Knight() 기본 생성자 호출" << endl;
    }

    Knight(int stamina) : Player(100) {
        /*
        선처리 영역이라고도 함
        - 여기서 Player(int hp) 생성자를 호출
        */


        _stamina = stamina;
        cout << "Knight(int stamina) 생성자 호출" << endl;
    }

    ~Knight() {
        cout << "Knight() 소멸자 호출" << endl;
    }
    /*
        후 처리 영역
        - 여기서 ~Player() 소멸자를 호출
        */

    // 재정의
    void Move() { cout << "Knight Move 호출" << endl; }
public:
    int _stamina;
};
```

```
class Mage : public Player {
public:

public:
    int _mp;
};

int main() {
    Knight k(100);

    k._hp = 100;
    k._attack = 10;
    k._defence = 5;
    //k._stamina = 50;

    //k.Move();
    // 밑 줄 과 같이 부모에서쓰던걸 호출할수도있긴함
    //k.Player::Move();
    //k.Attack();
    //k.Die();

    return 0;
}
```



선처리 영역과 후처리 영역의 순서를 잘 이해하자

은닉성

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

```
// 오늘의 주제 : 은닉성

// 객체지향 (OOP Object Oriented Programming)
// - 상속성
// - 은닉성 = 캡슐화
// - 다형성

// 은닉성(Data Hiding) = 캡슐화(Encapsulation)
// 몰라도 되는 것은 깔끔하게 숨기겠다!
// 숨기는 이유?
// - 1) 정말 위험하고 건드리면 안되는 경우
// - 2) 다른 경로로 접근하길 원하는 경우

// 자동차
// - 핸들
// - 페달
// - 엔진
// - 문
// - 각종 전기선

// 일반 구매자 입장에서 사용하는것?
// - 핸들/페달/문
// - 몰라도 됨(오히려 건드리면 큰일남)
// - 엔진, 각종 전기선

// public(공개적) protected(보호받는) private(개인의)
// - public : 누구에게나 공개. 실컷 사용하세요~
// - protected : 나의 자손들한테만 허락
// - private : 나만 사용할꺼! << class Car 내부에서만!

// 상속 접근 지정자 : 다음 세대한테 부모님의 유산을 어떻게 물려줄지?
// 부모님한테 물려받은 유산을 꼭 나의 자손들한테도 똑같이 물려줘야 하진 않음
// - public : 공개적 상속? 부모님의 유산 설계 그대로 (public -> publ
// - protected : 보호받은 상속? 내 자손들한테만 물려줄꺼야 (public ->
// - private : 개인적인 상속? 나까지만 잘쓰고 자손들한테는 안 물려줄꺼야

class Car {
public: // (멤버) 접근 지정자
```

```
    void MoveHandle() { }
    void PushPedal(){}
    void OpenDoor(){}

    void TurnKey() {
        // ...
        RunEngine();
    }

protected:
    void DisassembleCar() {} // 차를 분해
    void RunEngine() {} // 엔진을 구동
    void ConnectCircuit() {} // 전기선 연결

public:
    // 핸들
    // 페달
    // 엔진
    // 문
    // 각종 전기선
};

class SuperCar : private Car {
public:
    void PushRemoteController() {
        RunEngine();
    }
};

class TestSuperCar : public SuperCar {
public:
    void Test() {
        DissembleCar();
    }
};

// '캡슐화'
// 연관된 데이터와 함수를 논리적으로 묶어놓은 것
```

```
class Berserker {
public:

    int GetHp() { return _hp; }
    void SetHp(int hp) {
        _hp = hp;
        if (_hp <= 50)
            SetBerserkerMode();
    }
    // 사양 : 체력이 50 이하로 떨어지면 버서커 모드 발동(강해짐)

private:
    void SetBerserkerMode() {
        cout << "매우 강해짐!" << endl;
    }
private:
    int _hp = 100;
};

int main() {
    Car car;
    // 와! 너무 신기하다!
    // private 설정해주면 호출 못하게 됨
    //car.DisassembleCar();
    //car.RunEngine();

    Berserker b;

    b.SetHp(20);

    TestSuperCar car1;
    car1.PushRemoteController();

    return 0;
}
```

다형성

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

// 오늘의 주제 : 다형성

// 객체지향 (OOP Object Oriented Programming)
// - 상속성
// - 은닉성 = 캡슐화
// - 다형성

// 다형성(Polymorphism = Poly + morph) = 겉은 똑같은데, 기능이 다르
// - 오버로딩(Overloading) = 함수 중복 정의 = 함수 이름의 재사용
// - 오버라이딩(Overriding) = 재정의 = 부모 클래스의 함수를 자식 클래스

// 바인딩(Binding) = 묶는다
// - 정적 바인딩(Static Binding) : 컴파일 시점에 결정
// - 동적 바인딩(Dynamic Binding) : 실행 시점에 결정

// 일반 함수는 정적 바인딩을 사용
// 동적 바인딩을 원한다면? -> 가상 함수(virtual function)

// 그런데 실제 객체가 어떤 타입인지 어떻게 알고 알아서 가상함수를 호출해준
// - 가상 함수 테이블 (vftable)

// .vftable [] 4바이트(32) 8바이트(64)

// [VMove] [VDie]

// 순수 가상 함수 : 구현은 없고 '인터페이스'만 전달하는 용도로 사용하고 싶
// 추상 클래스 : 순수 가상 함수가 1개 이상 포함되면 바로 추상 클래스로 간
// - 직접적으로 객체를 만들 수 없게됨

class Player {
public:
    void Move() { cout << "Move Player!" << endl; }
    //void Move(int a) { cout << "Move Player(int)!" << endl;
    virtual void VMove() { cout << "VMove Player!" << endl; }
    virtual void VDie() { cout << "VDie Player!" << endl; }
```

```
    // 순수 가상 함수
    // 상속하는애가 VAttack을 반드시 구현해라
    virtual void VAttack() = 0;

public:
    int _hp;
};

class Kinght : public Player {
public:
    void Move() { cout << "Move Kinght!" << endl; }

    // 가상 함수는 재정의를 하더라도 가상 함수다!
    virtual void VMove() { cout << "VMove Kinght!" << endl; }
    virtual void VDie() { cout << "VDie Kinght!" << endl; }

    virtual void VAttack(){ cout << "VAttack Kinght!" << endl

public:
    int _stamina;
};

class Mage : public Player {
public:
    int _mp;
};

void MovePlayer(Player* player) {
    player->VMove();
    player->VDie();
}

int main() {
    //Player p;
    //MovePlayer(&p); // 플레이어는 플레이어다? Y
    //MoveKnight(&p); // 플레이어는 기사다? N
```

```
    Kinght k;
    //MoveKnight(&k); // 기사는 기사다? Y
    MovePlayer(&k); // 기사는 플레이어다? Y



    return 0;
}
```

초기화 리스트

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

// 오늘의 주제 : 초기화 리스트

// 멤버 변수 초기화? 다양한 문법이 존재

// 초기화 왜 해야할까? 귀찮다
// - 버그 예방에 중요
// - 포인터 등 주소값이 연루되어 있을 경우

// 초기화 방법
// - 생성자 내에서
// - 초기화 리스트
// - C++11 문법

// 초기화 리스트
// - 일단 상속 관계에서 원하는 부모 생성자 호출할 때 필요하다
// - 생성자 내에서 초기화 vs 초기화 리스트
// -- 일반 변수는 별 차이 없음
// -- 멤버 타입이 클래스인 경우 차이가 난다
// -- 정의함과 동시에 초기화가 필요한 경우 (참조 타입, const 타입)

class Inventory {
public:
    Inventory() { cout << "Inventory()" << endl; }
    Inventory(int size) { cout << "Inventory(int size)" << en
```

```
    ~Inventory() { cout << "~Inventory()" << endl; }
public:
    int _size = 10;
};

class Player {
public:
    Player(){}
    Player(int id) {}
};

// Is-A (Knight Is-A Player? 기사는 플레이어다) OK -> 상속관계

// Has-A (Knight Has-A Inventory? 기사는 인벤토리를 포함하고 있다 깆

class Knight : public Player {
public:
    Knight() : Player(1), _hp(100), _inventory(20), _hpRef(_h

        /*선처리 영역



        */

        _hp = 100;

        //_hpRef = _hp;
        //_hpConst = 100;;
    }

public:
    int _hp; // 쓰레기 값
    Inventory _inventory;

    int& _hpRef;
    const int _hpConst;
```

```
};


int main() {
    Knight k;

    cout << k._hp << endl;

    if (k._hp < 0) {
        cout << "Knight is Dead" << endl;
    }

    return 0;
}
```

연산자 오버로딩

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

// 오늘의 주제 : 연산자 오버로딩(Operator Overloading)

// 연산자 vs 함수
// - 연산자는 피연산자의 개수/타입이 고정되어 있음

// 연산자 오버로딩?
// 일단 [연산자 함수]를 정의해야 함
// 함수도 멤버함수 vs 전역함수가 존재하는것처럼, 연산자 함수도 두가지 방식

// - 멤버 연산자 함수 version
// -- a op b 형태에서 왼쪽으로 기준으로 실행됨 (a가 클래스여야 가능. a를
//-- 한계) a가 클래스가 아니면 사용 못함

// - 전역 연산자 함수 version
// -- a op b 형태라면 a, b 모두를 연산자 함수의 피연산자로 만들어준다

// 그럼 무엇이 더 좋은가? 그런거 없음. 심지어 둘 중 하나만 지원하는 경우도
// - 대표적으로 대입 연산자 (a=b)는 전역 연산자 version으로는 못 만든다
```

```
// 복사 대입 연산자
// - 대입 연산자가 나온김에 [복사 대입 연산자]에 대해 알아보자
// 용어가 좀 헷갈린다 [복사 생성자] [대입 연산자] [복사 대입 연산자]
// - 복사 대입 연산자 = 대입 연산자 중, 자기 자신의 참조 타입을 인자로 받

// 기타
// - 모든 연산자를 다 오버로딩 할 수 있는것은 아니다(:: . .* 이런건 안됨
// - 모든 연산자가 다 2개 항이 있는 것 아님. ++ --가 대표적(단항 연산자)
// - 증감 연산자 ++ --
// -- 전위형 (++a) operator++()
// -- 후위형 (a++) operator++(int)
class Position
{
public:
    Position operator+(const Position& arg)
    {
        Position pos;
        pos._x = _x + arg._x;
        pos._y = _y + arg._y;
        return pos;
    }

    Position operator+(int arg)
    {
        Position pos;
        pos._x = _x + arg;
        pos._y = _y + arg;
        return pos;
    }

    bool operator==(const Position& arg)
    {
        return _x == arg._x && _y == arg._y;
    }

    Position& operator=(int arg)
    {
```

```
        _x = arg;
        _y = arg;

        //Position* this = 내자신의 주소;
        return *this;
    }

    //복사 대입 연산자
    // [복사 생성자] [복사 대입 연산자] 등 특별대우를 받는 이유는,
    // 말 그대로 객체가 '복사'되길 원하는 특징 때문
    // TODO) 동적 할당 시간에 더 자세히 알아볼것
    Position& operator=(const Position& arg)
    {
        _x = arg._x;
        _y = arg._y;

        return *this;
    }

    Position& operator++()
    {
        _x++;
        _y++;
        return *this;
    }

    Position operator++(int)
    {
        Position ret = *this;
        _x++;
        _y++;
        return ret;
    }

public:
    int _x;
    int _y;
};
```

```
Position operator+(int a, const Position& b)
{
    Position ret;

    ret._x = b._x + a;
    ret._y = b._y + a;

    return ret;
}

// 이렇게 하면 원래의 왼쪽걸 오른쪽에 대입되게 할수도있음. 되게 위험함
// 따라서 막혀있다
//void operator=(const Position& a, int b)
//{
//  a._x = b;
//  a._y = b;
//}

int main()
{
    int a = 1;
    int b = 2;
    int c = (a = 3);
    int c = (a++);
    int c = (++a);
    int c = ++(++a);

    int c = a + 3.0f;

    Position pos;
    pos._x = 0;
    pos._y = 0;
    Position pos2;
    pos2._x = 1;
    pos2._y = 1;

    Position pos3 = pos + pos2;
```

```
    //pos3 = pos.operator+(pos2);

    Position pos4 = 1 + pos3;

    bool isSame = (pos3 == pos4);

    Position pos5;
    pos3 = (pos5 = 5);

    // (const Pos&)줘~       (Pos)복사값 줄께~
    pos5 = pos3++;

    ++(++pos3);

    return 0;
}
```