P1.S8

작성일시	@2024년 10월 7일 오후 7:59
강의 번호	C++ 언리얼
유형	강의
복습	☐
날짜	@2024년 10월 1일

동적 할당

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

// 오늘의 주제 : 동적 할당

// 메모리 구조 복습
// - 실행할 코드가 저장되는 영역 -> 코드 영역
// - 전역(global)/정적(static) 변수 -> 데이터 영역
// - 지역 변수/매개 변수 -> 스택 영역
// - 동적 할당 -> 힙 영역

// 지금까지 데이터 영역 / 스택 영역 이용해서
// 이런 저런 프로그램을 잘 만들어 왔다!
// 굳이 새로운 영역이 필요할까?

// 실제 상황)
// - MMORPG 동접 1명~5만명, 몬스터 1마리~500만 마리
// - 몬스터 생성 이벤트 -> 5분동안 몬스터가 10배 많이 나옴

// - 스택 영역
// 함수가 끝나면 같이 정리되는 불안정한 메모리
// 잠시 함수에 매개변수 넘긴다거나, 하는 용도로는 OK
// - 메모리 영역
// 프로그램이 실행되는 도중에는 '무조건' 사용되는
```

```
// 희망사항)
// - 필요할때만 사용하고, 필요없으면 반납할 수 있는!
// - 그러면서도 (스택과는 다르게) 우리가 생성/소멸 시점을 관리할 수 있는!
// - 그런 아름다운 메모리 없나? -> HEAP
// 동적할당과 연관된 함수/연산자 : malloc, free, new, delete, new[]

// malloc
// - 할당할 메모리 크기를 건내준다
// - 메모리 할당 후 시작 주소를 가리키는 포인터를 반환해준다(메모리 부족하
// free
// - malloc (혹은 기타 calloc, realloc 등의 사촌) 을 통해 할당된 영역
// - 힙 관리자가 할당/미할당 여부를 구분해서 관리

// new / delete
// - C++에 추가됨
// - malloc/free 함수! new/delete는 연산자(operator)

// new[] / delete[]
// - new가 malloc에 비해 좋긴한데~ 배열과 같이 N개의 데이터를 같이 할당

// malloc/free vs new/delete
// - 사용 편의성 -> new/delete 승!
// - 타입에 상관없이 특정한 크기의 메모리 영역을 할당받으려면? -> malloc

// 그런데 둘의 가장 가장 근본적인 차이는 따로 있음!
// new/delete는 (생성타입이 클래스일 경우) 생성자/소멸자를 호출해준다!!!

class Monster
{
public:
    Monster()
    {
        cout << "Monster()" << endl;
    }
    ~Monster()
    {
        cout << "~Monster()" << endl;
    }
```

```
public:
    int _hp;
    int _x;
    int _y;

};

Monster monster[500 * 10000];

int main()
{
    // 유저 영역 [메모장] [게임]
    // ---------------------------------
    // 커널 영역 (Windows 등의 핵심 코드)

    // 유저 영역) 운영체제에서 제공하는 API 호출
    // 커널 영역) 메모리 할당해서 건내줌
    // 유저 영역) 잘쓸게요~~

    // [    메모리를 할당할 때 큼지막하게 받아와서, 잘 나눠서씀      ]

    // C++ 에서는 기본적으로 CRT(C런타임 라이브러리)의 [힙 관리자]를 통
    // 단, 정말 원한다면 우리가 직접 API를 통해 힙을 생성하고 관리할 수도



    //그런데 잠깐! void* ?? 무엇일까
    // *가 있으니깐 포인터는 포인터 (주소를 담는 바구니) => OK
    // 타고가면 void, 아무것도 없다? => NO
    // 타고가면 void, 뭐가 있는지 모르겠으니까 너가 적당히 변환해서 사용하
    void* pointer = malloc(sizeof(Monster));

    Monster* m1 = (Monster*)pointer;
    m1->_hp = 100;
    m1->_x = 1;
    m1->_y = 2;

    // Heap Overflow
```

```
// - 유효한 힙 범위를 초과해서 사용하는 문제

free(pointer);
// 해제 안해주게 되면 메모리 누수

// free를 여러번 하게 되면
// Double free
//- 이건 대부분 그냥 크래시만 나고 끝난다


//m1->_hp = 100;
//m1->_x = 1;
//m1->_y = 2;
// Use-After-Free
// pointer는 계속 사용할 수 있는 상태여서
// 계속 건드릴 수 있음
// - 프로그래머 입장 : OMG 망했다!
// - 해커 입장 : 심봤다!

Monster* m2 = new Monster;
m2->_hp = 200;
m2->_x = 2;
m2->_y = 3;
delete m2;

Monster* m3 = new Monster[5];
m3->_hp = 200;
m3->_x = 2;
m3->_y = 3;

Monster* m4 = (m3 + 1);
m4->_hp = 200;
m4->_x = 2;
m4->_y = 3;

delete[] m3;
```

```
    return 0;
}
```

타입 변환 1,2

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

// 오늘의 주제 : 타입 변환

// malloc -> void*을 반환하고, 이를 우리가 (타입 변환)을 통해 사용했었

class Knight
{
public:
    int _hp;
};

class Dog
{
public:
    Dog()
    {

    }
    // 타입 변환 생성자
    Dog(const Knight& knight)
    {
        _age = knight._hp;
    }

    // 타입  변환 연산자
    operator Knight()
    {

    }
public:
    int _age = 1;
```

```
    int _cuteness = 2;
};

class BullDog : public Dog
{
public:
    bool _french;
};


int main()
{
    // ---------------타입변환 유형(비트열 재구성 여부)------------

    // [1] 값 타입 변환
    // 특징) 의미를 유지하기 위해서, 원본 객체와 다른 비트열 재구성
    {
        int a = 123456789; // 2의 보수
        float b = (float)a; // 부동소수점(지수 + 유효숫자)
        cout << b << endl;
    }

    // [2] 참조 타입 변환
    // 특징) 비트열을 재구성하지 않고, '관점'만 바꾸는 것
    // 거의 쓸일은 없지만, 포인터 타입 변환도 '참조 타입 변환' 동일한 룰을
    {
        int a = 123456789; // 2의 보수
        float b = (float&)a; // 부동소수점(지수 + 유효숫자)
        cout << b << endl;
    }

    // ---------------안전도 분류--------------------

    // [1] 안전한 변환
    // 특징) 의미가 항상 100% 완전히 일치하는 경우
    // 같은 타입이면서 크기만 더 큰 바구니로 이동
    // 작은 바구나 -> 큰 바구니로 이동 OK (업캐스팅)
    // ex) char -> short, short -> int, int -> _int64
```

```
    {
        int a = 123456789; // 2의 보수
        _int64 b = a; // 부동소수점(지수 + 유효숫자)
        cout << b << endl;
    }

    // [2] 불안전한 변환
    // 특징) 의미가 항상 100% 일치한다고 보장하지 못하는 경우
    // 타입이 다르거나
    // 같은 타입이지만 큰 바구니 -> 작은 바구니 이동 (다운캐스팅)
    {
        int a = 123456789; // 2의 보수
        float b = a;
        short c = a;
        cout << b << endl;
        cout << c << endl;
    }

    // ---------------프로그래머 의도에 따라 분류------------------

    // [1] 암시적 변환
    // 특징) 이미 알려진 타입 변환 규칙에 따라서 컴파일러 '자동'으로 타입
    {
        int a = 123456789;
        float b = a; // 암시적으로
        cout << b << endl;

    }

    // [2] 명시적 변환
    {
        int a = 123456789;
        int* b = (int*)a; //
        cout << b << endl;
    }

    // ---------------아무런 연관 관계가 없는 클래스 사이의 변환-----
```

```
    // [1] 연관 없는 클래스 사이의 '값 타입' 변환
    // 특징) 일반적으로 안 됨 (예외 : 타입 변환 생성자, 타입 변환 연산자
    {
        Knight knight;
        Dog dog = (Dog)knight;

        Knight knight2 = dog;
    }

    // [2] 연관없는 클래스 사이의 참조 타입 변환
    // 특징) 명시적으로는 OK
    {
        Knight knight;
        // 어셈블리 : 포인터 = 참조

        // [ 주소 ] -> [ Dog ]
        Dog& dog = (Dog&)knight;
        dog._cuteness = 12;
    }

    // ----------------상속 관계에 있는 클래스 사이의 변환-----------
    // 특징) 자식->부모 OK   /   부모->자식 NO
    // [1] 상속 관계 클래스의 값 타입 변환

    {
        // 실패
        //Dog dog;
        //BullDog bulldog = (BullDog)dog;

        BullDog bulldog;
        Dog dog = bulldog;
    }

    // [2] 상속 관계 클래스의 참조 타입 변환
    // 특징) 자식->부모 OK   /   부모->자식 (암시적NO) (명시적OK)
    {
        //Dog dog;
        // 통과는 시켜줌
```

```
        //BullDog& bulldog = (BullDog&)dog;

        // [ age cuteness french ]
        BullDog bulldog;
        Dog& dog = bulldog;
    }

    // 결론)
    // [값 타입 변환] : 진짜 비트열도 바꾸고~ 논리적으로 말이 되게 바꾸는
    // - 논리적으로 말이 된다? (ex. BullDog -> Dog) OK
    // - 논리적으로 말이 안 된다 (ex. Dog -> BullDog, Dog -> Knigh
    // [참조 타입 변환] : 비트열은 냅두고 우리의 '관점'만 바꾸는 변환
    // - 땡깡 부리면(명시적 요구) 해주긴 하는데, 말 안 해도 '그냥' (암시
    // -- 안전하다? (ex. BullDog -> Dog&) '그냥' (암시적으로) OK
    // -- 위험하다? (ex. Dog -> BullDog&)
    // --- 메모리 침범 위험이 있는 경우는 '그냥' (암시적으로) 해주진 않음
    // --- 명시적으로 정말 정말 하겠다고 최종 서명 하면 OK
    return 0;
}
```

복습

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

// 오늘의 주제 : 복습

class Item
{
public:
    Item()
    {
        cout << "Item()" << endl;
    }
    Item(const Item& item)
    {
        cout << "Item(const Item& item)" << endl;
    }
```

```
    ~Item()
    {
        cout << "~Item()" << endl;
    }

public:
    int _itemType = 0;
    int _itemDbId = 0;

    char _dummy[4096] = {}; // 이런 저런 정보들로 인해 비대해진
};

void TestItem(Item item)
{

}

void TestItemPtr(Item* item)
{

}

int main()
{
    // 복습
    {
        // Stack [ type(4) dbid(4) dummy(4096) ]
        Item item;

        // Stack [ 주소(4~8) ] -> Heap [ type(4) dbid(4) dummy
        Item* item2 = new Item();

        TestItem(item);
        TestItem(*item2);

        TestItemPtr(&item);
        TestItemPtr(item2);
```

```
        // delete를 빼먹으면 메모리누수(Memory Leak) -> 점점 가용 메
        delete item2;
    }

    // 배열
    {
        cout << "-----------------------------------" << endl

        // 진짜 아이템이 100개 있는 것 (스택 메모리에 올라와 있는)
        Item item3[100] = {};

        cout << "-----------------------------------" << endl

        // 아이템이 100개 있을까요?
        // 아이템을 가리키는 바구니가 100개. 실제 아이템은 1개도 없을 수
        Item* item4[100] = {};

        for (int i = 0; i < 100; i++)
            item4[i] = new Item();

        cout << "-----------------------------------" << endl

        for (int i = 0; i < 100; i++)
            delete item4[i];

        cout << "-----------------------------------" << endl
    }
    return 0;
}
```

타입변환4, 5

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

// 오늘의 주제 : 타입 변환 (포인터)
```

```
class Knight
{
public:
    int _hp;
};

class Item
{
public:
    Item()
    {
        cout << "Item()" << endl;
    }

    Item(int itemType) : _itemType(itemType)
    {
        cout << "Item(int itemType)" << endl;
    }

    Item(const Item& item)
    {
        cout << "Item(const Item& item)" << endl;
    }

    virtual ~Item()
    {
        cout << "~Item()" << endl;
    }

    virtual void Test()
    {
        cout << "Test Item" << endl;
    }

public:
    int _itemType = 0;
    int _itemDbId = 0;
```

```
    char _dummy[4096] = {}; // 이런 저런 정보들로 인해 비대해진
};

enum ItemType
{
    IT_WEAPON = 1,
    IT_ARMOR = 2,
};

class Weapon : public Item
{
public:
    Weapon() : Item(IT_WEAPON)
    {
        cout << "Weapon()" << endl;
        _damage = rand() % 100;
    }

    ~Weapon()
    {
        cout << "~Weapon()" << endl;
    }

    void Test()
    {
        cout << "Test Weapon" << endl;
    }

public:
    int _damage = 0;
};

class Armor : public Item
{
public:
    Armor() : Item(IT_ARMOR)
    {
        cout << "Armor()" << endl;
```

```
    }

    ~Armor()
    {
        cout << "~Armor()" << endl;
    }
public:
    int _defence = 0;
};

void TestItem(Item item)
{

}

void TestItemPtr(Item* item)
{
    item->Test();
}

int main()
{
    // 연관성이 없는 클래스 사이의 포인터 변환 테스트
    {
        // Stack [ 주소 ] -> Heap [ _hp(4) ]
        Knight* knight = new Knight();

        // 암시적으로는 NO
        // 명시적으로는 OK

        // Stack [ 주소 ]
        //Item* item = (Item*)knight;
        //item->_itemType = 2;
        //item->_itemDbId = 1;
        // %%이런 행위는 역적 행위다%%

        delete knight;
    }
```

```
// 부모 -> 자식 변환 테스트
{
    Item* item = new Item();

    // [ [ Item ] ]
    // [ _damage ]
    //Weapon* weapon = (Weapon*)item;
    //weapon->_damage = 10;
    // 역적이다 역적!!!
}

// 자식 -> 부모 변환 테스트
{
    Weapon* weapon = new Weapon();

    // 암시적으로도 된다!
    Item* item = weapon;

    // [가상함수 복습]
    TestItemPtr(item);

    delete weapon;
}

// 명시적으로 타입 변환할 때는 항상 항상 조심해야 한다!
// 암시적으로 될 때는 안전하다?
// -> 평생 명시적으로 타입 변환(캐스팅)은 안 하면 되는거 아닌가?

Item* inventory[20] = {};

srand((unsigned int)time(nullptr));
for (int i = 0; i < 20; i++)
{
    int randValue = rand() % 2;
    switch (randValue)
    {
    case 0:
```

```
            inventory[i] = new Weapon();
            break;
        case 1:
            inventory[i] = new Armor();
            break;
        }
    }

    for (int i = 0; i < 20; i++)
    {
        Item* item = inventory[i];
        if (item == nullptr)
            continue;

        if (item->_itemType == IT_WEAPON)
        {
            Weapon* weapon = (Weapon*)item;
            cout << "Weapon Damage :" << weapon->_damage << e
        }
    }

    // ************************** 매우 매우 매우 중요 **********

    for (int i = 0; i < 20; i++)
    {
        Item* item = inventory[i];
        if (item == nullptr)
            continue;


        /*if (item->_itemType == IT_WEAPON)
        {
            Weapon* weapon = (Weapon*)item;
            delete weapon;
        }
        else
        {
            Armor* armor = (Armor*)item;
```

```
            delete armor;
        }*/

        delete item;
    }

    // [결론]
    // - 포인터 vs 일반 타입 : 차이를 이해하자
    // - 포인터 사이의 타입 변환(캐스팅)을 할 때는 매우 매우 조심하자!
    // - 부모-자식 관계에서 부모 클래스의 소멸자에는 까먹지 말고 virtual
    // %면접 단골 질문%

    return 0;
}
```

얕은복사 vs 깊은복사 1

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

// 오늘의 주제 : 얕은 복사 vs 깊은 복사

class Pet
{
public:
    Pet()
    {
        cout << "Pet()" << endl;
    }
    ~Pet()
    {
        cout << "~Pet()" << endl;
    }
    Pet(const Pet& pet)
    {
        cout << "Pet(const Pet& pet)" << endl;
    }
};
```

```
class RabbitPet : public Pet
{

};

class RabbitPet : public Pet
{

};

class Knight
{
public:
    Knight()
    {
        _pet = new Pet();
    }
    Knight(const Knight& knight)
    {
        _hp = knight._hp;
        _pet = new Pet(*(knight._pet)); // 깊은 복사
    }
    Knight& operator=(const Knight& knight)
    {
        _hp = knight._hp;
        _pet = new Pet(*(knight._pet)); // 깊은 복사
        return *this;
    }
    ~Knight()
    {
        delete _pet;
    }
public:
    int _hp = 100;
    // 참조값을 들고있는 형태로 만들어주면 타입변환에 의해서 토끼펫이나 다
    Pet* _pet;
};
```

```
int main()
{
    Pet* pet = new Pet();


    Knight knight; // 기본 생성자
    knight._hp = 200;
    knight._pet = pet;

    Knight knight2 = knight; // 복사 생성자
    //Knight knight3(knight);

    Knight knight3; // 기본 생성자
    knight3 = knight; // 복사 대입 연산자

    // [복사 생성자] + [복사 대입 연산자]
    // 둘 다 안 만들어주면 컴파일러 '암시적으로' 만들어준다

    // 중간 결론) 컴파일러가 알아서 잘 만들어준다?
    // 수고하세요~ 다음 주제 넘어갈까요? << NO

    // [ 얕은 복사 Shallow Copy ]
    // 멤버 데이터를 비트열 단위로 '똑같이' 복사 (메모리 영역 값을 그대로
    // 포인터는 주소값 바구니 -> 주소값을 똑같이 복사 -> 동일한 객체를 가

    // Stack : Knight1 [ hp 0x1000 ] -> Heap 0x1000=Pet[  ]
    // Stack : Knight2 [ hp 0x1000 ] -> Heap 0x1000=Pet[  ]
    // Stack : Knight3 [ hp 0x1000 ] -> Heap 0x1000=Pet[  ]

    // delete _pet; 이 세번 호출되는 상황


    // [ 깊은 복사 Deep Copy ]
    // 멤버 데이터가 참조(주소) 값이라면, 데이터를 새로 만들어준다 (원본
    // 포인터는 주소값 바구니 -> 새로운 객체를 생성 -> 상이한 객체를 가리

    // Stack : Knight1 [ hp 0x1000 ] -> Heap 0x1000=Pet[  ]
```

```
    // Stack : Knight2 [ hp 0x2000 ] -> Heap 0x2000=Pet[  ]
    // Stack : Knight3 [ hp 0x3000 ] -> Heap 0x3000=Pet[  ]

    return 0;
}
```

얕은복사 vs 깊은복사 2

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

// 오늘의 주제 : 얕은 복사 vs 깊은 복사

class Pet
{
public:
    Pet()
    {
        cout << "Pet()" << endl;
    }
    ~Pet()
    {
        cout << "~Pet()" << endl;
    }
    Pet(const Pet& pet)
    {
        cout << "Pet(const Pet& pet)" << endl;
    }
    Pet& operator=(const Pet& pet)
    {
        cout << "operator=(const Pet& pet)" << endl;
        return *this;
    }
};

class Player
{
public:
```

```
    Player()
    {
        cout << "Player()" << endl;
    }

    // 복사 생성자
    Player(const Player& player)
    {
        cout << "Player(const Player& player)" << endl;
        _level = player._level;
    }

    // 복사 대입 연산자
    Player& operator=(const Player& player)
    {
        cout << "operator=(const Player& player)" << endl;
        _level = player._level;
        return *this;
    }
public:
    int _level = 0;
};

class Knight : public Player
{
public:
    Knight()
    {

    }
    Knight(const Knight& knight) : Player(knight), _pet(knigh
    {
        cout << "Knight(const Knight& knight)" << endl;

        _hp = knight._hp;
        _stamina = knight._stamina;
    }
    Knight& operator=(const Knight& knight)
```

```
    {
        cout << "operator=(const Knight& knight)" << endl;

        Player::operator=(knight);

        _pet = knight._pet;

        _hp = knight._hp;
        return *this;
    }
    ~Knight()
    {

    }
public:
    int _hp = 100;
    int _stamina = 100;
    Pet _pet;
};

int main()
{
    Pet* pet = new Pet();


    Knight knight; // 기본 생성자
    knight._hp = 200;
    knight._level = 99;

    //cout << "------------- 복사 생성자 ---------------" << end
    //Knight knight2 = knight; // 복사 생성자
    //Knight knight3(knight);

    Knight knight3;

    cout << "------------- 복사 대입 연산자 ---------------" << 
    knight3 = knight; // 복사 대입 연산자
```

```
    // 실험)
    // - 암시적 복사 생성자 Steps
    // 1) 부모 클래스의 복사 생성자 호출
    // 2) 멤버 클래스의 복사 생성자 호출
    // 3) 멤버가 기본 타입일 경우 메모리 복사 (얕은 복사 Shallow Copy)

    // - 명시적 복사 생성자 Steps
    // 1) 부모 클래스의 기본 생성자 호출
    // 2) 멤버 클래스의 기본 생성자 호출

    // - 암시적 복사 대입 연산자 Steps
    // 1) 부모 클래스의 복사 대입 연산자 호출
    // 2) 멤버 클래스의 복사 대입 연산자 호출
    // 3) 멤버가 기본 타입일 경우 메모리 복사 (얕은 복사 Shallow Copy)

    // - 명시적 복사 대입 연산자 Steps
    // 1) 알아서 해주는거 없음

    // 왜 이렇게 혼란스러울까?
    // 객체를 '복사' 한다는 것은 두 객체의 값들을 일치시키려는 것
    // 따라서 기본적으로 얕은 복사(Shallow Copy) 방식으로 동작

    // 명시적 복사 -> [모든 책임]을 프로그래머한테 위임하겠다는 의미

    return 0;
}
```

캐스팅 4총사(면접 단골)

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

// 오늘의 주제 : 캐스팅 (타입 변환)


class Player
{
public:
```

```
    virtual ~Player() {}
};

class Knight : public Player
{
public:
};

class Archer : public Player
{
public:
};

void PrintName(char* str)
{
    cout << str << endl;
}

class Dog
{

};

// 1) static_cast
// 2) dynamic_cast
// 3) const_cast
// 4) reinterpret_cast

int main()
{
    // static_cast : 타입 원칙에 비춰볼 때 상식적인 캐스팅만 허용해준다
    // 1) int <-> float
    // 2) Player* -> Knight* (다운캐스팅) << 단, 안정성 보장 못함

    int hp = 100;
    int maxHp = 200;
    float ratio = static_cast<float>(hp) / maxHp;
```

```
    // 부모 -> 자식 자식 -> 부모, 즉 상관관계가 있어야함(다이나믹캐스트도
    Player* p = new Archer();
    Knight* k1 = static_cast<Knight*>(p);

    // dynamic_cast : 상속 관계에서의 안전 형변환
    // RTTI (RunTime Type Information)
    // 다형성을 활용하는 방식
    // - virtual 함수를 하나라도 만들면, 객체의 메모리에 가상 함수 테이
    // - 만약 잘못된 타입으로 캐스팅을 했으면, nullptr 반환 *********
    // 이를 이용해서 맞는 타입으로 캐스팅을 했는지 확인하기에 유용하다
    Knight* k2 = dynamic_cast<Knight*>(p);



    // const_cast : const를 붙이거나 떼거나~
    PrintName(const_cast<char*>("Rookiss"));

    // reinterpret_cast
    // 가장 위험하고 강력한 형태의 캐스팅
    // 're-interpret' : 다시~간주하다/생각하다
    // - 포인터랑 전혀 관계없는 다른 타입 변환 등
    __int64 address = reinterpret_cast<__int64>(k2);

    Dog* dog1 = reinterpret_cast<Dog*>(k2);

    void* p = malloc(1000);
    Dog* dog2 = reinterpret_cast<Dog*>(p);

    return 0;
}
```