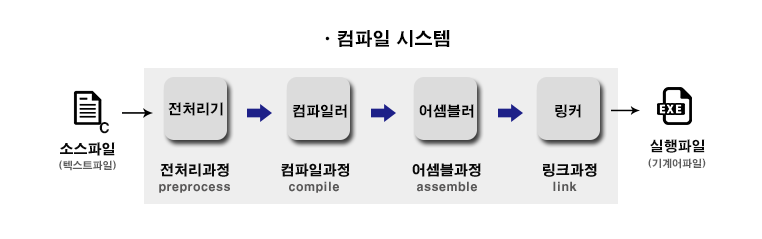
고수준언어 c / c++ 등 아래 과정들을 exe파일 생성



어셈블리언어 어셈블러로 기계어 변환후 link과정을 거쳐 exe파일 생성

Float(4바이트) : 부호(1) 지수(8) 유효숫자(23) = 32bit 사용

Float double은 근사값이다

값이 커질수록 오차가 커진다

UTF-8

알파벳, 숫자 1바이트 ( ASCII 와 동일한 번호)

유럽 지역의 문자 2바이트

한글, 한자 등 3바이트

UTF-16

알파벳, 숫자, 한글, 한자 등 거의 대부분 문자 2바이트

매우 예외적인 고대 문자만 4바이트(무시해도됨)

UTF-16을 사용하는 wchar\_t

Ex) wchar\_t wch = L’안’;

++a / a++의 차이

B = ++a / b = a++

전자는 a에 1을 더한값이 b에 들어가고 후자는 b에 a값이 들어가고 a를 1증가

\_\_int64 a, b = 0;

a = b + 3;

int a, b = 0;

a = b + 3;

위에는 rax레지스터 사용 / 아래는 eax레지스터사용

X64 / x84 어떤걸로 디버그를 하느냐에 따라 달라짐

32비트로 디버그하면 8바이트 정수를 써도 eax레지스터사용 단 구성이 조금 달라짐

>>

비트 오른쪽으로 이동. 부호 비트 이동 안시킴

<<

비트 왼쪽으로 이동. 부호 비트도 포함

텍스트이(가) 표시된 사진

자동 생성된 설명

\

Int hp = 77777;

Short hp2 = hp; // int hp의 위쪽 16비트가 짤림

Float hp3 = hp; // 정밀도 차이가있어 데이터 손실 가능

int hp = -1;

unsigned int hp4 = hp; // unsigned int의 최대값이뜸 int에서 -1은 모든 비트를 1로 킨건데

unsigned int관점에서 모든 비트를 1로 키면 최대값임 부호비트가 없기때문

상수 const, enum, #define -> enum 0순위

Struct PlayerInfo

{

Short hp;

Int attack;

Char defence;

};

* 메모리 크기 2+4+1 = 7같지만 아님 padding(빈메모리)가 들어감 그럼 저건 몇이냐?

// 자료의 배치순서, 가장 큰 바이트값에 영향을 받음

2byte+padding(2) + 4 + 1 + padding(3) = 12byte

But

Short hp; char defence; int attack 순으로 저장됐을 때

2+padding(1) + 1 + 4 = 8byte임

Char 변수 3개만 저장됐을 때

-> 3byte 가장큰 바이트값이 1byte이기때문

\_\_int64 데이터가 추가됐을 때

1+padding(3) + 4 + (1 + padding(7)) + 8 = 24byte

1 + 4 + padding(3) -> 이렇게 저장되지 않음 (하나하나 끊겨야함)

함수 오른쪽에 const 리턴형에 const

Const int getA();

Int getA() const;

const상수 선언 const int a;

struct PlayerInfo

{

Int attack;

Int dence;

Int hp;

}

PlayerInfo player;

Player의 시작주소는 int attack의 주소와 같다

Int attack의 주소 +4에 defence, 그 +4에 hp가있다 연속으로 3개의 변수 선언과 비슷

PlayerInfo CreatePlayer();

Void CreatePlayer(PlayerInfo\* player); 의 차이

1번째 방법은 많은 복사가 이루어져 비용을 많이먹는다

2번째 방법은 원본값을 그대로 수정하는것이기 때문에 1번재 방법보다는 비용이 덜 드는 2번째 방법을이용하는 것이 좋다

1. Void CreatePlayer(PlayerInfo player); 값 전달 방식
2. Void CreatePlayer(PlayerInfo\* player); 주소 전달 방식

그냥 값을 수정하지않고 읽기만 한다면 둘쪽다 문제가없음

Int number = 1;

// 4바이트 정수형 바구니 그 바구니의 이름을 number로

// number의 값을 읽거나, number 값을 넣을때는

//찰떡같이 알아듣고 해당주소(stack, data, heap)에 가서 해당주소의 값을 수정하거나 읽는다

Int\* ptr = &number;

Ptr : 4바이트 정수형 번수의 주소를 담는 바구니(주소값 : 8byte(64bit cpu))

Int& red = number;

//로우레벨(어셈블리) 관점에서 실제 작동 방식은 int\*와 같음

//사용방식이 int\*와 다르게 int처럼 사용

//c++관점에서는 number라는 바구니에 또 다른 이름(별명)을 부여한 것

//number라는 바구니에 ref라는 다른 이름을 지어줄게

//앞으로 ref바구니에서 뭘 꺼내거나 넣으면

//실제 number바구니에서 꺼내거나 넣을게

실제 작동방식이 int\*와 같은데 왜 int&을 만드냐?

장점

1. 구조체의 경우 Int.hp, int.attack ‘->’이 아닌 ‘.’연산자로 보기편안함
2. 보기도 편안하고 int\* 효율도 좋아서 사용한다고 함. 일석이조의 방식인데 단점도있음

단점

1. 너무 편리하기 때문에 코드가 몇만줄이 되고 함수도 많아질 때 원본을 넘긴다는 것을 모를수가있음 포인터는 원본을 넘긴다는 것이 잘 보임 수정하면 안되는 데이터를 맘대로 고쳐버릴수가 있음
2. 참조하는 대상이 꼭 있어야함 2번째 별명을 만드는 느낌이기 때문에 포인터는 nullptr을 넣어줄수있지만 참조타입은 대상이 꼭 존재해야만 한다.

포인터 vs 참조

결론? Team by team..

1. 읽기만 하는경우 const type&
2. nullptr일수 있는경우 pointer
3. 그 외 일반적으로 ref (명시적으로 호출할 때 define OUT를 붙인다라던가 원본을 수정하다는것을 알려줌)

마음대로 고치지 못치지 못하게 const를 되도록이면 붙여줌

포인터로 사용하던걸 참조로 넘겨주려면?

PlayerInfo\* player = nullptr;

Void CreatePlayer(PlayerInfo& player);

CreatePlayer(player) -> 불가

CreatePlayer(\*player); 이렇게 해줘야함

참조로 사용하던걸 포인터로 넘겨주려면?

PlayerInfo& player2 = player;

CreatePlayer(&player);

Const \* 전에붙이냐 \* 후에 붙이냐

별 뒤에 붙였을 때

PlayerInfo\* const info;

//Info = &otherInfo;

* 주소값 수정이 불가능

별 이전에 붙였을 때

Const PlayerInfo\* info;

PlayerInfo const\* info; 둘다 똑같지만 이 방법이 일반적임

//Info->hp = 10000;

* 원본 데이터값을 수정할수없음

Const char\* str = “Hello World”;

// .rdata 영역의 어떤 주소에 “Hello World” 값을 넣어주고 그 주소를 str에 저장

멤버변수 네이밍 컨벤션

m\_hp

\_hp

mHp

크게 3가지중 하나로

\_hp가 좋을것같음

함수명 기능을 포함하며 제일앞 대문자 Attack

멤버함수에서 멤버변수를 사용하는건 this->가 생략되는거임

\_hp = 0

this->\_hp = 0;

복사생성자

Knight(const Knight& knight)

{ \_hp = knight.\_hp; …

Knight k1(k2)

Knight k1 = k2;

둘다 복사생성자를 불러옴

Knight k1

k1 = k2;

이건 기본 생성자를 불러온후에 k2정보를 복사하는것

인자를 1개받는 생성자 : 타입 변환 생성자

Knight(int hp) { … }

Knight k1;

k1 = 1; -> == Knight(1)랑 동일함 컴파일러가 알아서해줌

이걸 의도하는 경우가 거의 없음 그럼에도 불구하고 사용하는 경우가 있기 때문에

명시적으로 함수를 사용하라고 explicite를 붙여줌 explicit Knight(int hp) {…}

생성은 부모 자식순 생성

Player() -> Knight()

엄밀히 말하면 조금 다르긴함 Knight() … {…} …->선처리구역 :\_hp(hp)같이 :붙이는곳

Knight생성자 함수를 호출하고 함수 내부를 처리하기전에 선처리구역에서 Player의 생성자를 호출하기 때문에 결과론적으로는 Player생성자가 먼저 호출되고 Knight생성자 함수가 호출되는것처럼 보임

소멸은 자식 부모순 소멸

~Knight() -> ~Player();

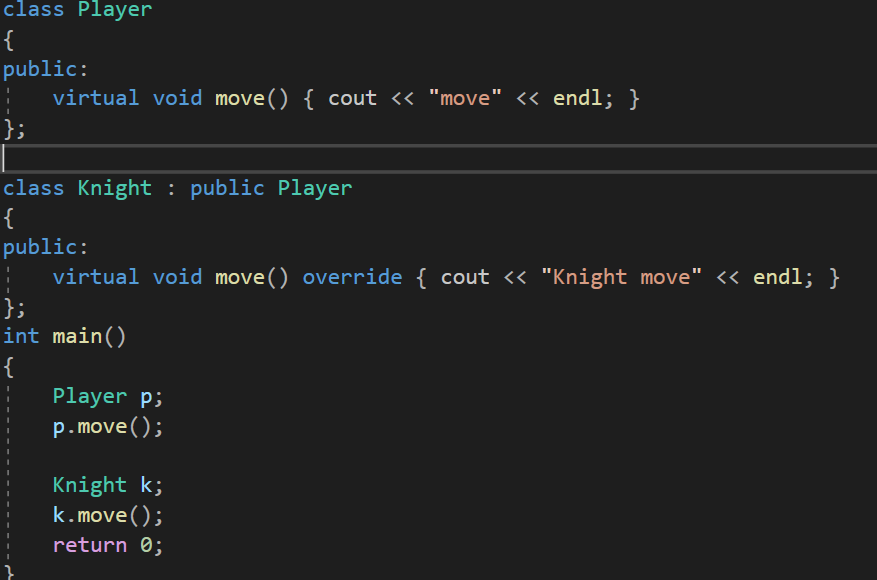
~Player()는 ~Knight()의 후처리영역에서 실행된다고 한다.

바인딩 : 컴파일러가 기계어로 바꾸는데 함수와 어떤 주소를 매칭해서 어떤 함수를 실행시킨다

일반함수 -> 정적 바인딩 : 컴파일 시점에 결정

virtual func -> 동적 바인딩 : 실행 시점에 결정

사용방법



실제 객체가 어떤 타입인지 어떻게 알고 알아서 가상함수를 호출해준걸까?

* 가상 함수 테이블이 만들어짐 (vftable) 어셈블러 까보면 나옴
* 객체 생성자 함수 호출할 때 선처리 영역에서 자신이 vftable에서 어떤 함수를 호출할지 선택함

순수 가상함수 : 구현은 없고 ‘인터페이스’만 전달하는 용도로 사용하고 싶을경우

Player -> knight, mage, archer

플레이어는 knight, mage, archer 3타입중에 하나로만 존재해야만 하는데 player라는 타입으로는 존재해서는 안됨 이걸 막기위해 순수 가상함수를 추가해 해당 클래스를 추상클래스로 만들면됨

virtual void VAttack() = 0; // modern c++ virtual void VAttack() abstract;

부모에서 저렇게 선언만 해주면 자식들은 저 함수를 꼭 구현해야만함

추상 클래스 : 순수 가상함수가 1개 이상 포함되면 해당 클래스는 추상 클래스로 간주

추상 클래스로 만들어진 객체는 직접적으로 만들수 없음

main에서 Player p; 이렇게 객체를 만들수 없음

vftable이 실제 객체 첫번째 주소 오프셋에 박혀있음

초기화 방법

* 생성자 내에서 Knight() { \_hp = hp; }
* 초기화 리스트 사용 Knight(int hp) :\_hp(hp) { … }
* c++11 문법 사용 \_hp = 100; // 선언과 동시에 초기화

초기화 리스트에서 초기화하는 것이 베스트

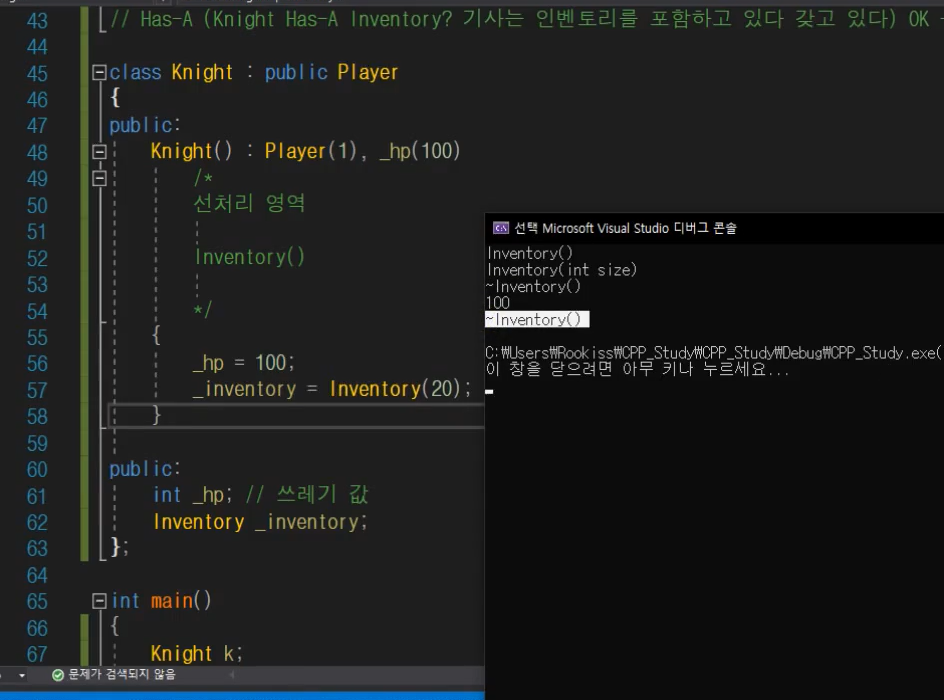
예외의경우 – 정의함과 동시에 초기화가 필요한 경우 (참조 타입, const 타입)

멤버변수로 int& \_hpRef, const int MaxHp 선언한다면 -> 무조건 초기화 리스트에서 해줘야함

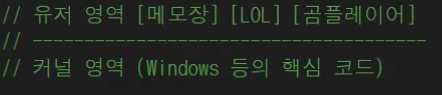
설계할 때 고려할 것

Is-A? Knight Is A Player (기사는 플레이어인가? ok ->상속관계)

Has\_A? Knight Has A Inventory ( 기사는 인벤토리를 포함하고 있는가? ok -> 포함관계



\_inventory를 Knight의 멤버변수로 가지고있으면 Knight생성자 함수 선처리영역에서 \_inventory의 생성자함수를 호출하는데 \_inventory를 Knight생성자 함수 내부에서 초기화하면 생성자 함수가 2번 호출되는것을 볼수있음 소멸자가 뭉쳐서 뜨지 않는 이유는 다른 생성자 함수를 호출하면서 먼저 호출한 생성자함수에 대한 소멸자함수를 호출하는것임



windows, mac os, android 등의 핵심 코드는 커널 영역에서 실행중임

유저영역에서 동적 할당을하면 운영체제에서 제공하는 API를 호출해서 메모리를 달라고 요청하는것임

C++에서는 기본적으로 CRT(C런타임 라이브러리)의 [힙 관리자]를 통해 힙 영역 사용

힙 관리자를 통해 많은 힙 영역을 받고 요청이 오면 일부분 떼어서 사용하는식임

단, 정말 원한다면 우리가 직접 API를 통해 힙을 생성하고 관리할 수도 있음.  
(MMORPG 서버 메모리 풀링)

void\*

* 주소를 담는 바구니 ok
* 타고가면 void 뭐가 있는지 모르겠으니까 너가 적당히 변환해서 사용해라 ok

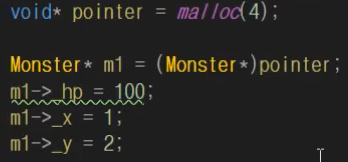
void\* ptr = malloc(1000);

Monster\* m1 = (Monster\*)ptr;

m1->\_hp = 100; …

malloc

* 할당한 메모리 크기를 건내준다.
* 메모리 할당 후 시작 주소를 가리키는 포인터를 반환한다 ( 메모리 부족 NULL)



Heap Overflow

* 유효한 힙 범위를 초과해서 사용하는 문제
* free / delete를 안해주고 계속 메모리 할당을 요청한다면 메모리 누수(Memory Leak)가 발생

Use-After-Free

* free를 해주고 m1을 다시 사용할 때 m1사용하던 메모리에 중요한 정보가 들어있다면 밀어버릴수가 있음 조심해야함

malloc/free vs new/delete

* malloc/free는 함수 new/delete는 연산자(operator)
* 사용 편의성 new/delete
* type에 상관없이 특정 크기의 메모리를 할당받고 싶다면 malloc/free
* @@new/delete는 생성 타입이 class일경우 생성자/소멸자를 호출해준다

타입 변환

* [1] 값 타입 변환
* 특징) 의미를 유지하기 위해, 원본 객체와 다른 비트열 재구성

int a = 123456789; // 2의 보수로 숫자 표현

float b = (float)a; // 부동소수점(지수 + 유효숫자)로 표현

최대한 의미를 유지할수 있기위해 비슷한 값으로 조정해주어 오차가 생김

* [2] 참조 타입 변환
* 특징) 비트열을 재구성하지 않고, ‘관점’ 만 바꾸는 것
* 거의 쓸일은 없지만, 포인터 타입 변환도 ‘참조 타입 변환’과 동일한 룰을 따르니까 일석이조로 공부하자!

int a = 123456789;

float b = (float&)a;

a의 주소에 들어있는 값과 b의 주소에 들어있는 값은 같음

그러나, cout << b; 를 하면 a와 전혀 다른 값이 뜨는데

b에 들어있는 값을 부동소수점으로 분석을 하기 때문에 값을 바라보는 관점이 다르다는 것을 알수있음

안전도에 따른 변환

* [1] 안전한 변환
* 특징) 의미가 항상 100% 완전히 일치하는 경우
* 같은 타입이면서 크기만 더 큰 바구니로 이동 // 작은바구니 -> 큰바구니 ( 업캐스팅)
* ex) char -> short, short -> int, int -> \_\_int64
* [2] 불안전한 변환
* 특징) 데이터가 손실이 일어날수있다.
* 타입이 다르거나 // 같은 타입이지만 큰 바구니 -> 작은 바구니 (다운캐스팅)

프로그래머 의도에 따른 변환

* [1] 암시적 변환
* 특징) 이미 알려진 타입 변환 규칙에 따라서 컴파일러가 ‘자동’으로 타입 변환

int a = 123456789;

float b = a; ->암시적으로 변환 딱히 요청하지 않아도 변환

* [2] 명시적 변환

int a = 123456789;

int\* b = (int\*)a; // (int\*)가 없으면 오류 의도하고 싶다면 명시적으로 변환해야함

연관 없는 클래스 사이의 값 타입 변환

타입 변환 생성자, 타입 변환 연산자

Knight knight;

Dog dog = (Dog)knight;

Knight knight2 = dog;

연관 없는 클래스 사이의 참조 타입 변환

명시적으로 ok;

Knight knight;

Dog& dog = (Dog&)knight;

상속 관계인 클래스 사이의 변환

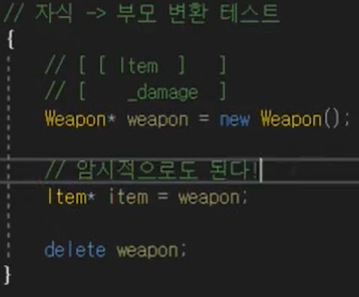
* [1] 상속 관계 클래스의 값 타입 변환
* 특징) 자식->부모 ok, 부모->자식 no

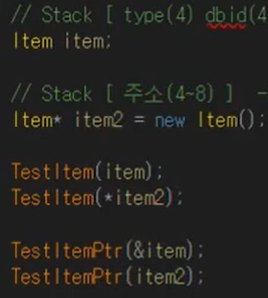
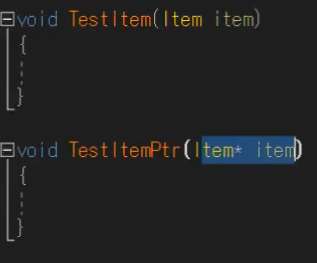
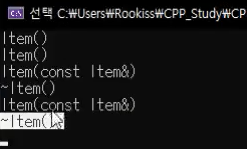
BullDog bulldog;

Dog dog = bulldog;

bulldog에서 dog의 정보만 dog에 들어감

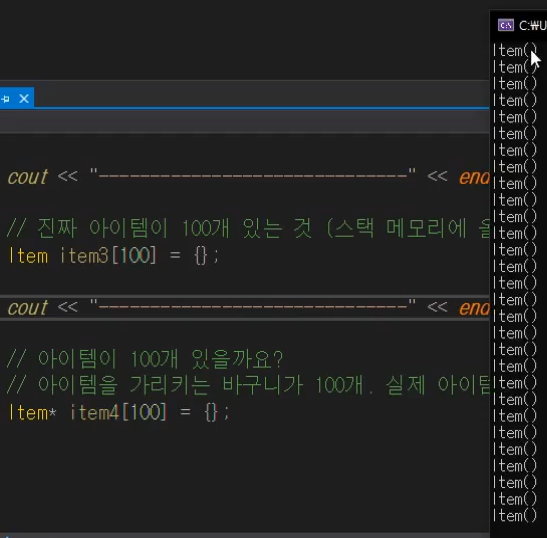
* [2] 상속 관계 클래스의 참조 타입 변환
* 특징) 자식->부모 ok, 부모->자식 암시적No 명시적ok

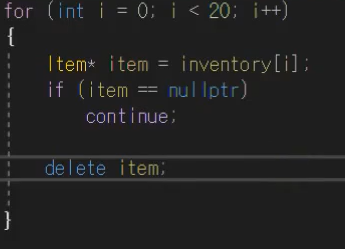
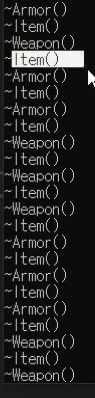
 

TestItem함수의 경우 매개변수인 Item item의 복사 참조 생성자 함수와 소멸자를 호출함

TestItemPtr함수의 경우 생성자를 생성하지않고 주소만받고 이용하는식임



소멸자에 virtual 해야하는 이유

무기와 아머가 item에 상속받고 있고 item에 무기와 아머를 생성하여 보관하고 있는데

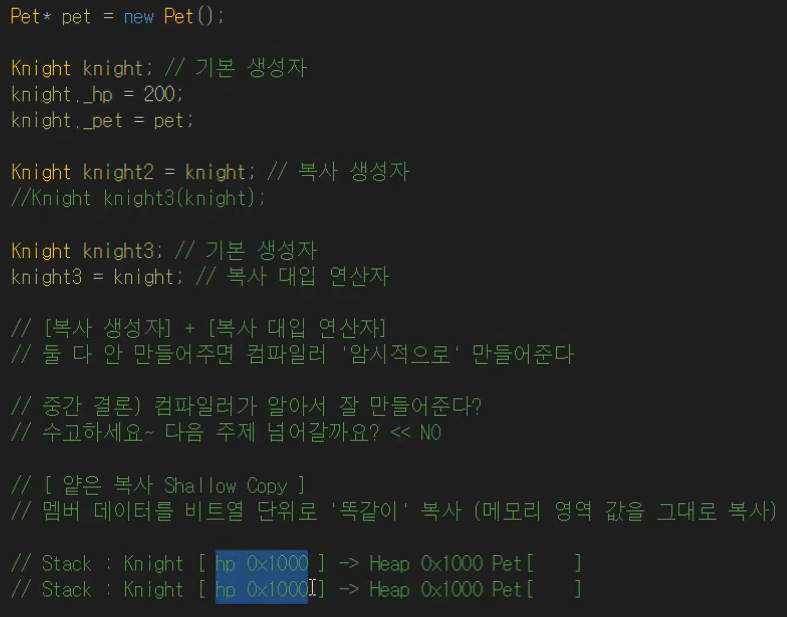
delete item을 한다면 item의 소멸자함수만 호출한다.

여기서는 Heap Memory에 누수가 발생함.

결론 : 해당 객체에 대한 소멸자를 제대로 호출 안해준다면 Memory Leak 발생!

얕은복사 vs 깊은복사

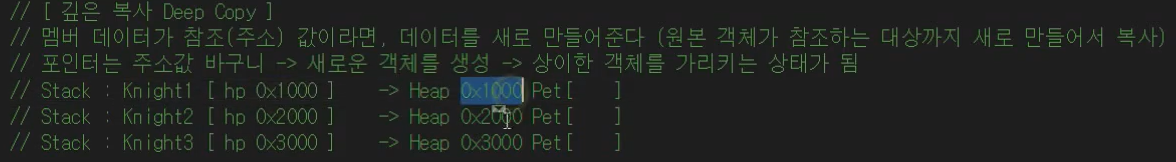
* 얕은 복사
* 특징) 멤버 데이터를 비트열 단위로 ‘똑같이’ 복사 (메모리 영역 값을 그대로 복사)



Knight2 와 Knight3의 Pet\* \_pet멤버변수가 Knight의 \_pet을 똑같이 가리키고 있음

3개의 나이트가 똑같은 pet을 가리키고있음. 주소값을 똑같이 복사하고 있기 때문에 동일한 객체를 가리키는 상태임. 만약 Knight의 소멸자 함수에서 pet을 delete한다면 같은 pet을 3번이나 delete해주는것임

* 깊은 복사
* 특징) 멤버 데이터가 참조(주소) 값이라면, 데이터를 새로 만들어준다. (원본 객체가 참조하는 대상까지 새로 만들어서 복사)



컴파일러가 자동으로 해주는 복사생성자 , 복사 대입 연산자로 생성한 객체는 얕은 복사임.



명시적으로 복사 생성자나 복사 대입 연산자를 정의하고 주소값을 가지는 멤버변수에 새로운 객체를 생성해서 그 펫의 정보를 복사할 펫의 정보로 복사한다. 즉 펫의 정보는 같지만 주소갚은 다름

상속 관계가 있는 얕은 복사 vs 깊은복사 (c++ 강의 얕은 복사 vs 깊은 복사 #2 보면됨)

* 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)
* 명시적 복사 생성자 steps
* 1) 부모 클래스의 기본 생성자 호출
* 2) 멤버 클래스의 기본 생성자 호출

자식 클래스에서 명시적인 복사 생성자 없이(암시적) 부모 클래스에서만 복사 생성자를 만들어줬을경우 Knight k2(k1) 이러면 player와 pet의 복사생성자를 호출함 그러나 Knight에서 자신만의 명시적인 복사 생성자를 만들어준다면 player와 pet의 기본 생성자만 호출하고 복사생성자는 호출하지 않으며 Knight본인만의 복사 생성자만을 호출함. Knight만 복사되고 player와 pet의 정보는 짤리는 결과가 발생됨.



이렇게 Player와 pet의 복사생성자를 호출 해줘야 전체적인 복사를 할수있음.

암시적 복사 대입 연산자나 명시적 복사 대입 연산자 위와 동일함

궁금한 것

Inline 함수

왜 변수들을 하나씩 선언할때는 주소값이 32떨어져서 선언이 되는데

배열로 선언하면 바로 다음 주소에 값들이 들어가는지

클래스안에서의 static 멤버변수

Static 변수