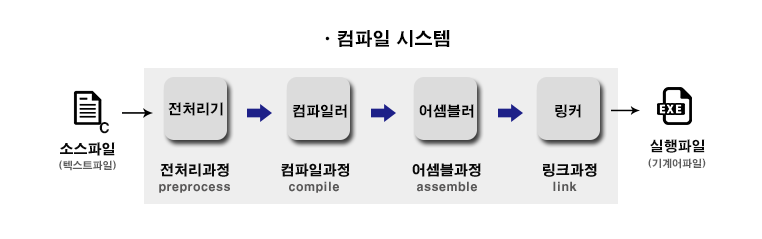
고수준언어 c / c++ 등 아래 과정들을 exe파일 생성



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();

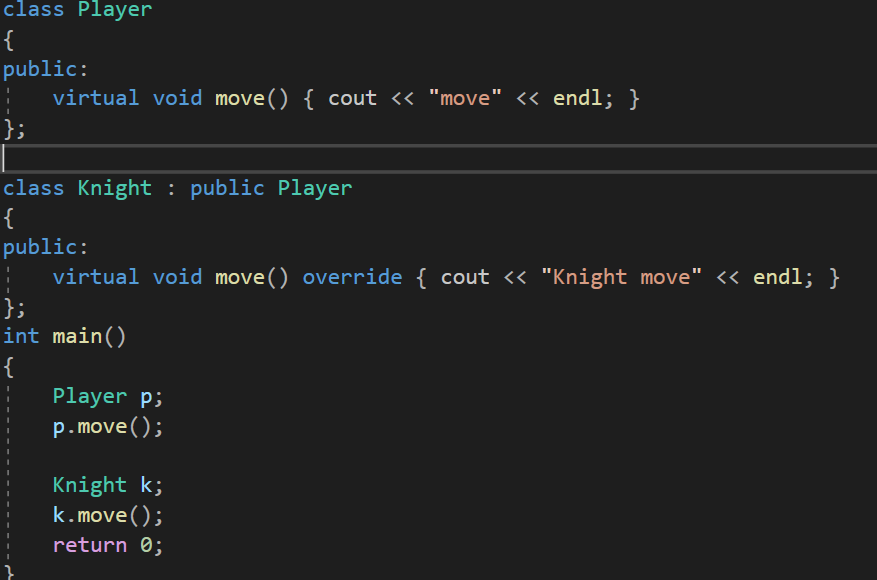
~Player()는 ~Knight()의 후처리영역에서 실행된다고 한다.

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

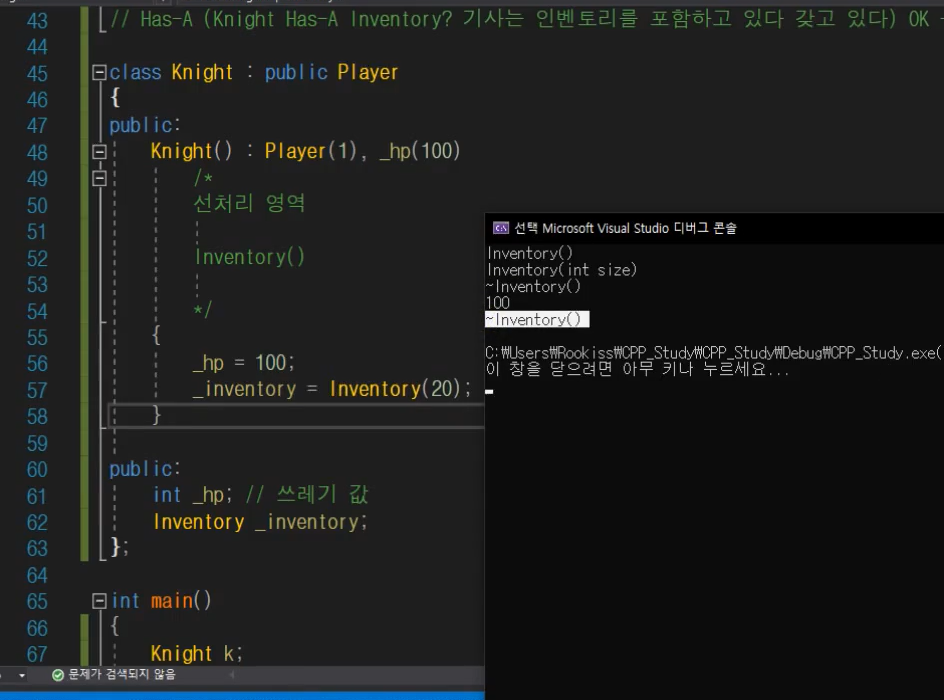
예외의경우 – 정의함과 동시에 초기화가 필요한 경우 (참조 타입, const 타입)

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

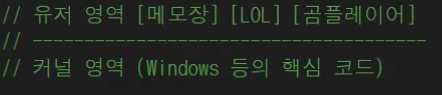
설계할 때 고려할 것

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

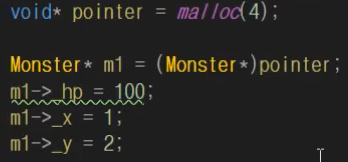
void\* ptr = malloc(1000);

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

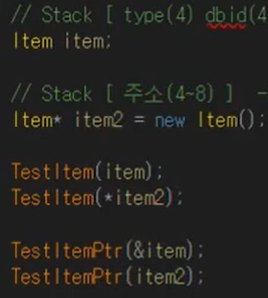
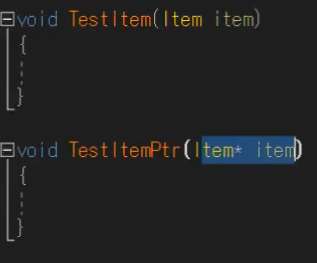
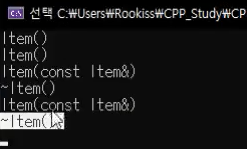
BullDog bulldog;

Dog dog = bulldog;

bulldog에서 dog의 정보만 dog에 들어감

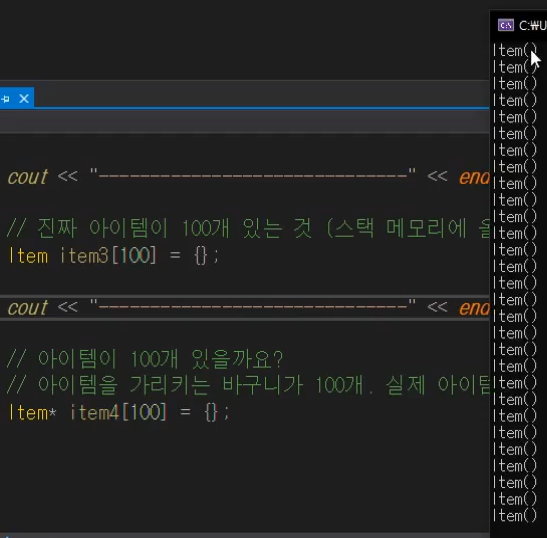
* [2] 상속 관계 클래스의 참조 타입 변환
* 특징) 자식->부모 ok, 부모->자식 암시적No 명시적ok

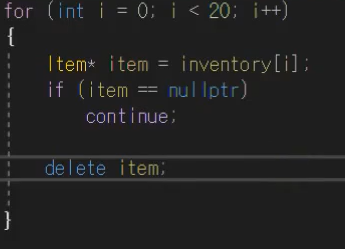
 

TestItem함수의 경우 매개변수인 Item item의 복사 참조 생성자 함수와 소멸자를 호출함

TestItemPtr함수의 경우 생성자를 생성하지않고 주소만받고 이용하는식임



소멸자에 virtual 해야하는 이유

무기와 아머가 item에 상속받고 있고 item에 무기와 아머를 생성하여 보관하고 있는데

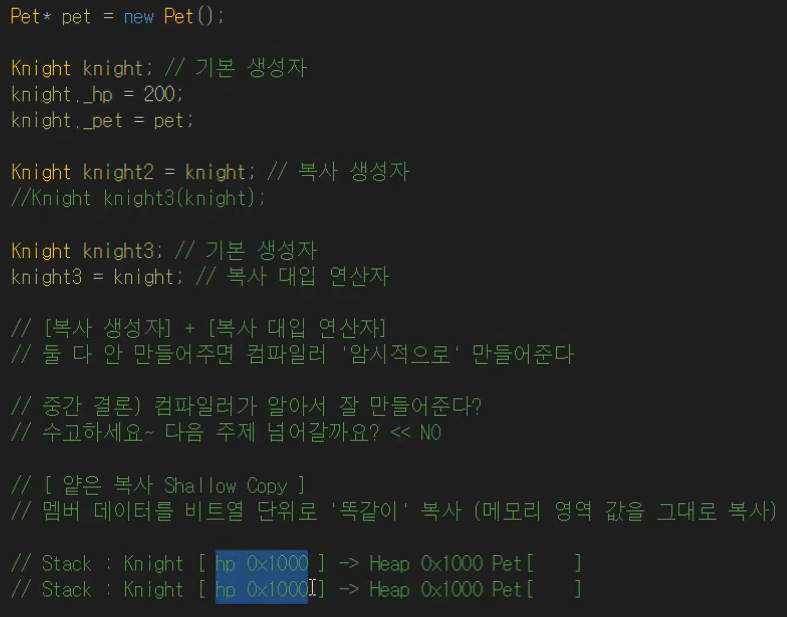
delete item을 한다면 item의 소멸자함수만 호출한다.

여기서는 Heap Memory에 누수가 발생함.

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

얕은복사 vs 깊은복사

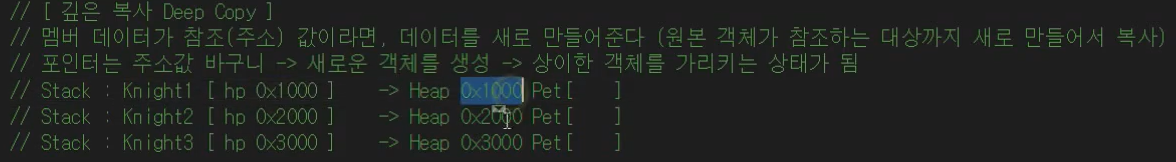
* 얕은 복사
* 특징) 멤버 데이터를 비트열 단위로 ‘똑같이’ 복사 (메모리 영역 값을 그대로 복사)



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



전방 선언을 하는 이유

동적 할당을 할때는 해당 타입의 바이트 수를 알고있어야 한다.

직접 정의한 클래스의 변수를 멤버 변수로 가지고 있을때는 해당 클래스의 바이트 수를 모르기 때문에 #indlue “Monster.h> 로 헤더파일 안에서 다른 헤더파일을 include 해줘야한다.

그러나 직접 정의한 클래스의 포인터 변수를 멤버 변수로 가지고 있을때는 32bit cpu에서는 4byte 64bit cpu 에서는 8byte로 확정 되기 때문에 바이트 수를 정확하게 계산할 수 있다.

전방 선언을 해주는 이유는 해당 타입이 정말로 존재하는지 컴파일러는 모르기 때문에 에러를 만들어주는데 해당 타입이 존재한다고 컴파일러에게 알려주기 위해서 전방선언을 한다.

직접 정의한 클래스의 포인터 변수를 멤버 변수로 가지고 있고 cpp파일에서 해당 포인터 변수를 사용할때는 그 클래스의 설계도(클래스)를 알지 못하면 해당 주소로 접근을 해도 어떻게 설계되었는지 몰라 .\_hp .\_attack등의 주소를 알수없어 수정하지못한다.

함수 포인터

typedef int DATA; -> 일반 타입 정의 방법

typedef int(int a, int b) FUNC\_TYPE

* 문법이 조금 달라짐
* typedef int (FUNC\_TYPE)(int a, int b); -> 함수 정의 방법

typedef int (FUNC\_TYPE)(int a, int b); -> original

typedef int (FUNC\_TYPE)(int, int);

using FUNC\_TYPE = int(int a, int b); //modern c++

using FUNC\_TYPE = int(int, int);

함수의 타입을 정의하다, 변수 명은 없어도 된다.

typedef int (FUNC\_TYPE)(int a, int b);

int type의 매개변수 2개를 받으며 int type을 return하는 함수이다.

typedef int (FUNC\_TYPE2)(int a);

int type의 매개변수 1개를 받으며 int type을 return하는 함수이다.

int Add(int a, int b) { return a+b; }

int result = Add(1, 2);

어셈블리어를 까보면 Add함수를 Call하는데 Add함수의 주소로 jmp한다.

* 함수의 이름은 함수의 시작주소이다. 함수도 주소를 가지고 있다.

FUNC\_TYPE\* fn;

fn = Add; -> 2개의 int 인자를 받고 1개의 int를 반환하는 signature는 해당 타입에 대입가능

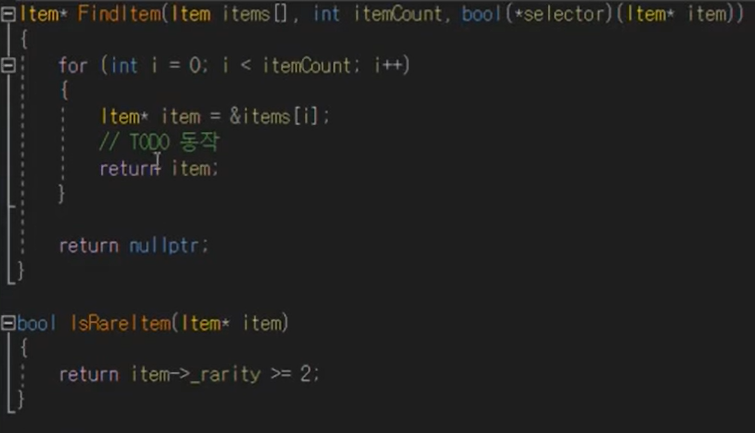
fn에는 Add함수의 주소값이 들어간다.

int result = Add(1, 2) == fn(1,2)

int result = (\*fn)(1,2); 라고도 표현한다. 어셈블리어 까봐도 정확히 똑같음

함수 포인터는 \*(접근 연산자)가 붙어도 함수 주소이다.

사용법

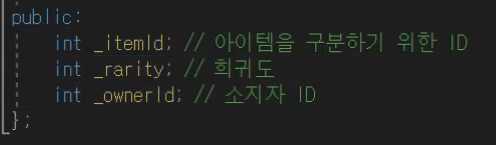


어떤 함수의 인자로 어떤 동작을 넣을수 있다.

bool 타입을 반환하고 item\* 를 인자로 받는 함수로 item의 조건을 찾는다.

typedef bool(ITEM\_SELECTOR)(Item\* item);

Item\* FindItem(Item item[], int itemCount, ITEM\_SELECTOR\* selector) 로 바꿔도 동일하다.

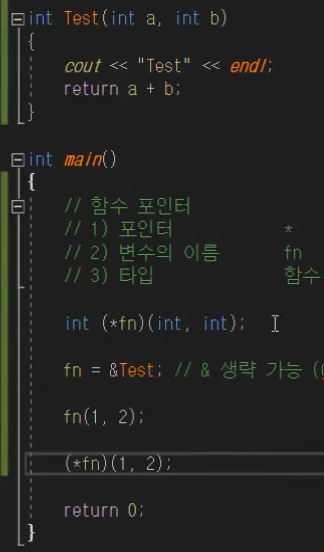


인자로 넣는 함수만 바꿔줘도 \_rarity 로만 아이템을 찾을 수 있을뿐 아니라 다른 id나 ownerId로도 찾을수 있다.

FindItem(items, items.length(), IsRareItem);

FindItem(items, items.length(), IsIdItem);

FindItem(items, items.length(), IsOwnerIdItem);

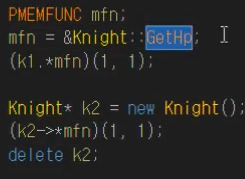


전혀 문제없이 돌아감 fn을 저렇게 정의도 가능

위 문법으로는 [전역 함수 / 정적 함수]만 담을수 있음 (호출 규약이 동일한 애들)

멤버함수 포인터는 조금 다름

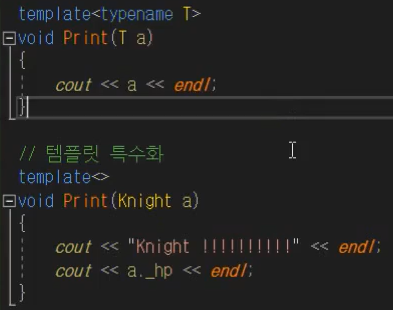




상속 관계가 아닌 다른 클래스의 멤버함수는 안됨

상속 관계의 경우 넣을수 있긴 하지만 고급주제이므로 필요할 때 검색을 통해 쓰길바람.

함수 템플릿



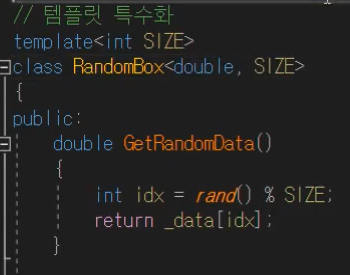
함수 템플릿 특수화 : 특수한 예외사항을 두고싶다면 템플릿 특수화를 통해 Knight를 넣었을때는 아래 버전의 함수가 실행되게 할수있음.

클래스 템플릿



random으로 int 나 float 등의 랜덤값을 얻을 수 있는 클래스

다른 버전의 클래스가 만들어지는 것이므로 복사 연산자나 복사 생성자를 통해 대입이 불가능함



클래스 템플릿 특수화 ( 위와 비교 바람)

STL(Standard Templete Library)

Vector // 동적 배열

텍스트이(가) 표시된 사진

자동 생성된 설명

Reserve(), capacity(), size(), resize(), clear(), push…

Reserve() : 여유분을 잡는 것 capacity를 늘림

처음부터 조금 크게 여유분을 잡으면 복사 비용을 줄일 수 있어 좀 더 효율적이다

Clear() : size를 지워버리지만 capacity를 줄이지는 않음

Capacity도 비워버릴려면 vector<int> v; vector<int>().swap(v); 으로 빈 벡터를 v와 swap하여 v를 초기상태로 만들수있음.

Iterator ( c++ vector #2~3 )

텍스트이(가) 표시된 사진

자동 생성된 설명

포인터느낌

STL / 자료구조 / 알고리즘 / 면접 단골 질문

* 컨테이너에서 중간 삽입 / 삭제
* 처음 / 끝 에서의 삽입 / 삭제
* 임의 접근 ( Random Access )

Vector

* 중간 삽입 / 삭제 (BAD) : 중간에 낑겨넣으면서 뒤에있는 값들을 하나씩 뒤로 전부 복사해줘야함 ex) 5를 낑겨넣었을 때 : 0 1 2 3 4 5 [] [] -> 0 1 5 2 3 4 5 [] 삭제도 비슷하게 생각하면됨
* 처음 / 끝 에서의 삽입 / 삭제 : 처음값의 삽입 삭제는 위와 똑 같은 이유로 비용이 많이듬 Bad, 그러나 끝에서의 삽입 삭제의 경우 그 값만 삽입하거나 삭제하면 되므로 간단함 GOOD
* 임의 접근 : ex) Vector<int> v(10); v[2] = 3; 등의 접근이 가능함.

C++ STL 시퀀스 컨테이너 Vector List Deque

C++ STL List

vector 와 list의 차이?

list의 동작 원리

List

* 중간 삽입 / 삭제 : GOOD
* 처음, 끝 의 삽입 / 삭제 : GOOD
* 임의 접근 ( read, write) : 불가능

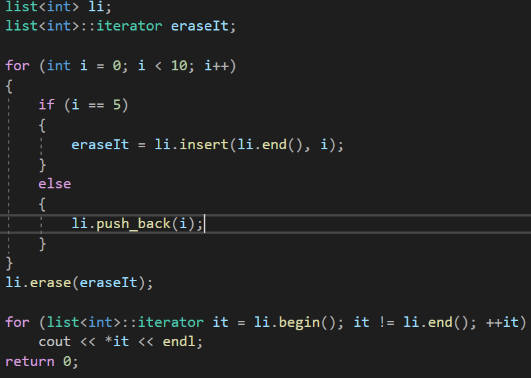
어떨 때 vector를 쓰고 어떨 때 list를 쓰는지 내 자신만의 기준점이나 프레임워크가 있다면 굉장히 좋을것 같은 설계를 할 때 어떤 자료구조를 쓸지 그 이유를 알수있기때문

데이터들이 처음과 중간의 삽입 삭제가 빈번하다면 list를 쓰는 것이 맞을 듯

어떤 자료구조를 쓸지 정할 때 고민해야할 것 [[[[[삽입 삭제 정렬 탐색]]]]]] 비용

list는 탐색 비용이 좋지않을 듯 싶음

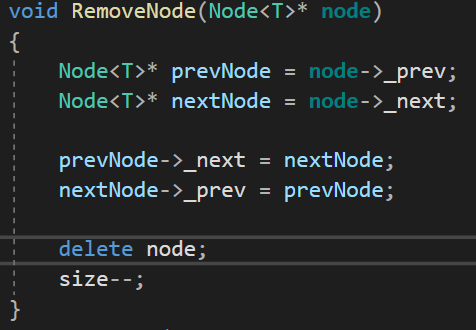
ex) 100개의 데이터를 들고있는 list에서 50번째에 있는 데이터를 삭제해주세요 -> 비용이 많이듬 ?? 중간에 있는 데이터는 삭제가 빠르다면서요? -> 50번째 데이터를 iterator로 기억하고 있지 않다면 굉장히 느림. -> 설계 할 때 이 데이터는 언젠가 삭제될것이라는 것을 알고 데이터를 삽입할때 iterator로 기억하고있어야함.

 이렇게 기억하고있어야함

리스트에서 데이터를 직접 정의한 클래스로 가지고있고 그 클래스 에서 주소를 가지고있는 데이터가 있을 때

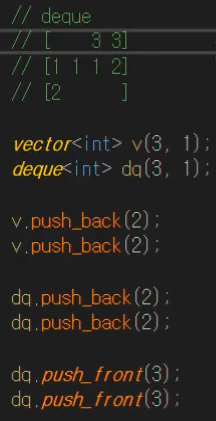
리스트에서 노드를 remove했을 때 해당 데이터의 delete는 어떻게 해주는건가?

템플릿을 특정 템플릿으로 만들어 removeNode함수에서 해당 데이터의 소멸자함수를 호출해주는것은 어떨까?

 그냥 removeNode함수에서 delete node를 하면 node의 소멸자 함수를 호출하면서 T타입이 클래스여도 해당 클래스의 소멸자 함수도 같이 호출되면서 메모리를 줄일수있음.

Deque

* vector와 마찬가지로 배열 기반으로 동작
* 다만 메모리 할당 정책이 다르다

통이 꽉차면 같은 사이즈의 새로운 통을 만드는 것

주소가 이어져 있지 않으며 주소를 까보면 통끼리 전혀 다른 주소에 위치하고있다.

* 중간 삽입 / 삭제 : (BAD, BAD)
* 처음, 끝 삽입 / 삭제 : (GOOD , GOOD) push\_front, push\_back 지원
* 임의 접근 : Deque<int> d(3, 1); d[0] = 2; 가능

C++ STL 연관 컨테이너 Map, Set, Multimap, Multiset

vector나 list같은 시퀀스 컨테이너가 있는데 왜 이런 연관 컨테이너를 써야할까?

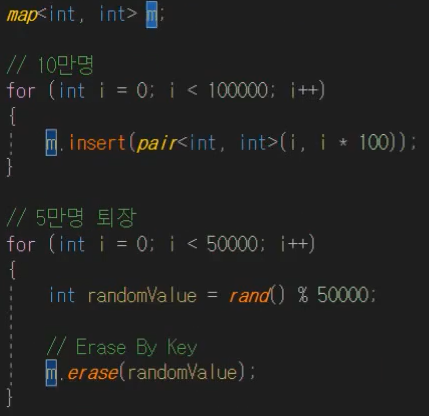
시퀀스 컨테이너의 치명적인 단점이있다. 원하는 조건에 해당하는 데이터를 빠르게 찾을 수 없다.

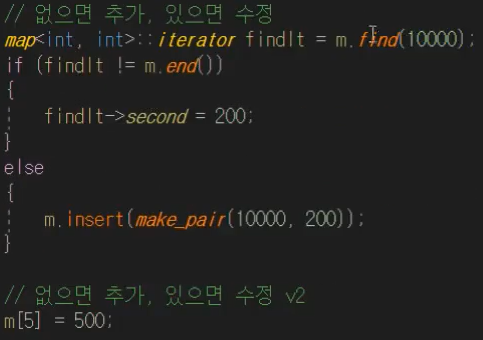
ex) id=2만번인 플레이어가 id=3만번인 플레이어를 공격하였다. 라는 패킷을 서버에 보내왔다.

서버는 id=2만번과 3만번인 플레이어가 있는지 확인하고 핵등을 위한 여러 조건을 체크한 후에 3만번 플레이어의 hp를 깎는다. 이런경우 2만번 플레이어와 3만번 플레이어를 탐색하기 위해 전체 플레이어를 탐색할 수 없기 때문에 최소한의 비용만으로 2만번과 3만번 플레이어를 찾기위해 연관 컨테이너를 쓰게되었다.

Map ( 자료구조 시간때 상세하게 작성할 예정)

* 균형 이진 트리 ( AVL)
* 노드 기반
* 어떤 특정 조건에 따라 정렬되어 있음





사용 방법:

map<T1, T2> m; // T1 : KEY, T2 : VALUE

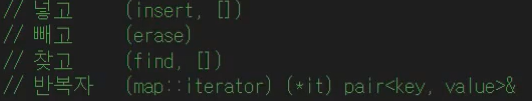
map에 데이터를 넣을때는 pair라는 타입으로 insert한다.

키값이 있는데 키값을 통해 탐색을 할수있다.

pair라는 타입도 key값과 value값을 넣는다.

key값이 이미 존재하는데 중복되게 값을 넣거나 지우면 뒤에 넣은 값은 무시된다

[] 연산자가 사용가능한데 m[5] = 500; 이라 한다면 5번키에 데이터가 없으면 추가하고 있으면 값만 수정한다.



Set

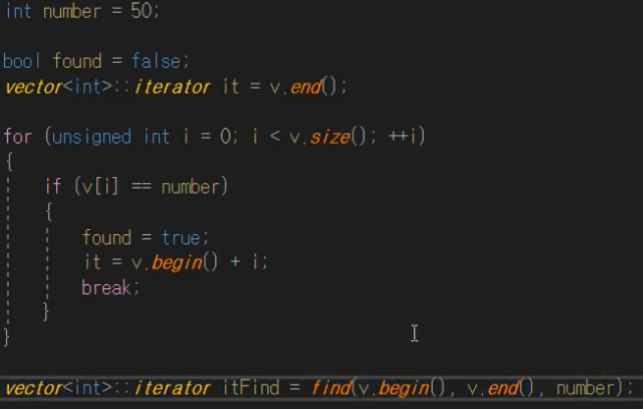
* map에서 value값만 빠졌다 생각하면됨
* map에서 key값과 value값을 동일하게 생각한다면 굳이 map을 안쓰고 set를 쓰면됨

Multimap

* key값이 중복되게 넣을 수 있는 map이라 생각하면됨(많이 쓸일은 없다고함)
* 같은 key에 다른 value를 넣어도됨

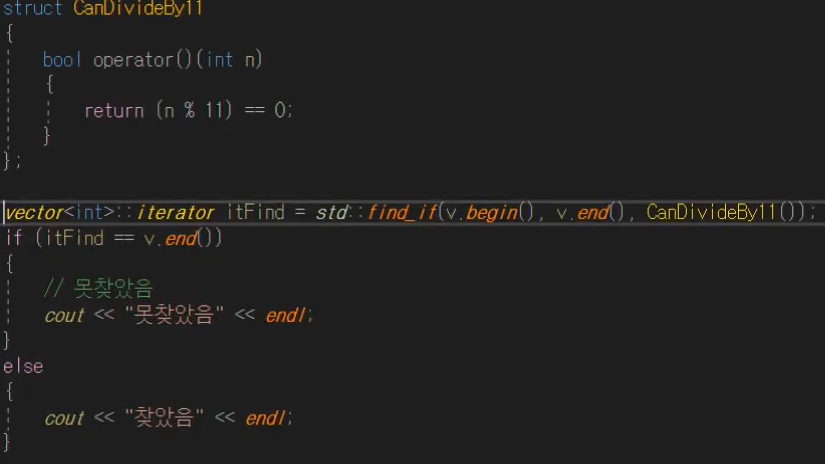
표준 C++에서 제공하는 알고리즘

find

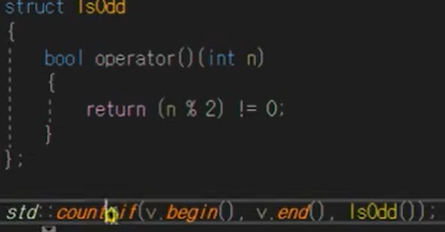


find\_if

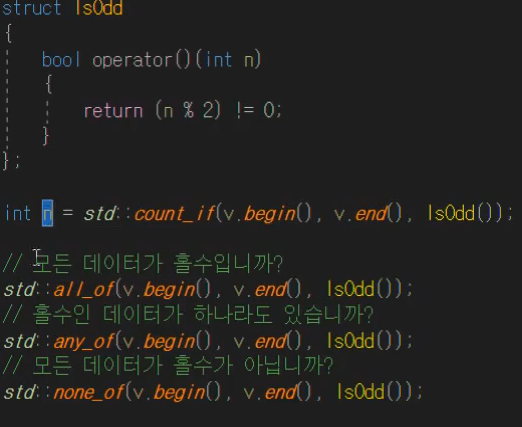
find\_if 함수도 있는데 마지막 인자로 함수 객체를 넣어줘서 어떤 조건을 만족하는 데이터를 가져올 수 있음 find함수와 비슷하게 사용하면되고 마지막 인자로 람다 문법을 이용하면 간단하게 뽑아낼수있음



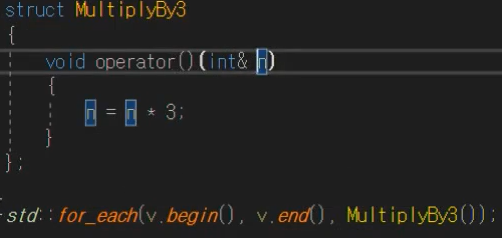
count, count\_if 함수 ex) 벡터에서 홀수인 데이터가 몇 개인지 구하기



all\_of, any\_of, none\_of 함수 ex) 벡터에서 홀수인 데이터 구하기

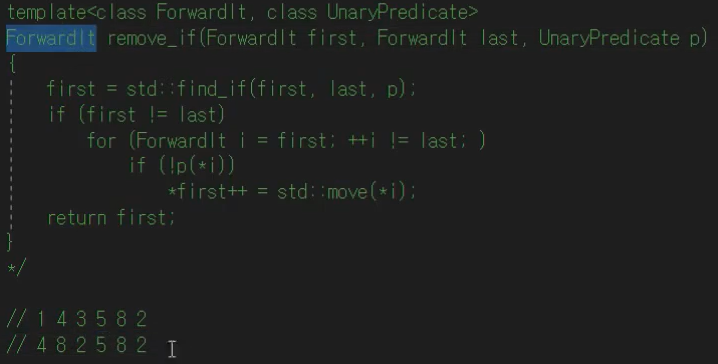


for\_each 함수 ex) 모든 벡터에 3곱하기



remove, remove\_if 조심히 써야함

벡터에서 clear나 erase 굉장히 조심해야함 왜? 모르면 vector영상 다시보기



1 4 3 5 8 2 에서 홀수만 찾고 나머지는 삭제하고 싶을 때 remove\_if(v.begin(), v,end(), idOdd())라고 하고 출력하면 4 8 2 5 8 2가 출력이 되는데 remove\_if함수는 삭제를 시켜주는게 아니라 조건에 만족하는 데이터들을 찾아주고 더 이상 사용하지 않아도 되는 iterator를 return 시켜줌 즉 4 8 2를 찾아주고 5에 위치한 부분을 리턴시켜주는거임 v.end()에 쓰레기 값이 들어있는것과 비슷하게 생각하면됨

modern c++11

audo

* 기본 const 와 & 무시함 조심해야함

int& ref = a;

auto t1 = ref -> int t1 = ref랑 똑같음

auto& t1 = ref 로 해줘야 t1이 int&타입이 됨.

{} 연산자

Nullptr

오버로딩 문제때문에

Using // 왜 typedef가 있는데 using을 쓰느냐? 밑에 이유가 있음.

텍스트이(가) 표시된 사진

자동 생성된 설명

이제 typedef는 안씀 using 쓰면됨

Enum class

그냥 enum과의 차이점?

* 그냥 enum은 unscoped enum임 즉, 범위없는 enum으로 해당 이름으로 enum형을 만들면 그 이름으로는 다른 enum을 만들수없음. None으로 처음 열거형을 만드는 경우가 많은데 PT\_NONE이라든지 열거형의 약자를 따와서 하나만 존재할수 있게함. Enum class는 해당클래스 내부에서만 유효하며 다른 열거형과 이름이 겹칠수있음.
* 암묵적인 변환이 금지됨. 단순히 0, 1, 2.. 등등이 아님.

둘중 하나 골라서 사용해도 상관없음

Delete ( 삭제된 함수 )

* 정의를 하지않은 비공개(private) 함수를 만드는것

텍스트이(가) 표시된 사진

자동 생성된 설명 ex) 대입 연산자 금지시킬수있음.

Override, final ( virtual func 과 관련이 있음 )

* Override : 부모의 가상함수를 재정의 한다는 keyword // 함수 오른쪽에 붙임
* Final : 더 이상 자식들에게 override 하게 하고싶지 않을 때 사용.

오른값 참조

lvalue, rvalue

텍스트이(가) 표시된 사진

자동 생성된 설명

2번째 함수는 Knight클래스의 생성자함수를 불러와서 그 객체를 아무리 사용해봤자 함수가 끝나면 없어지기 때문에 사용 자체가 불가했다. 하지만 3번째 함수처럼 const만 붙이면 읽기 용도로만 사용하겠다 라고 해서 일회용 Knight 객체를 보내줄수있다.

일회용 객체를 받을 수 있지만 3번째 함수는 매개변수로 const를 붙인 객체를 받는다면 그 객체의 멤버 변수들을 수정할 수 없는 const함수들만 호출할수있다. 이 단점도 보완해주는 키워드가 있는데 오른값 참조이다.

Knight&& knight를 매개변수로 받는 함수를 만든다면 lvalue들은 받지 않고 rvalue들만 받으며 객체의 멤버 변수에 접근할수있다. 그치만 어차피 함수가 끝나면 없어질 일회용 객체가 왜 필요한걸까?

원본은 날려도 된다는 의미가 있다. 이동 대입 연산자로 많이쓰인다. 원본의 데이터를 복사본으로 뻇어온다는 느낌이다. Void operator=(Knight&& knight) {… 얕은 복사 후 원본 nullptr} => 쪽쪽 빨아먹고 원본 버림.

K2 = static\_cast<Knight&&>(k1);

K2 = std::move(k1);

위 두 방식처럼 사용하면 된다.

오른값 참조 사용예시

포인터는 포인터인데 이 세상에서 딱 하나만 존재해야하는 포인터

Std::unique\_ptr



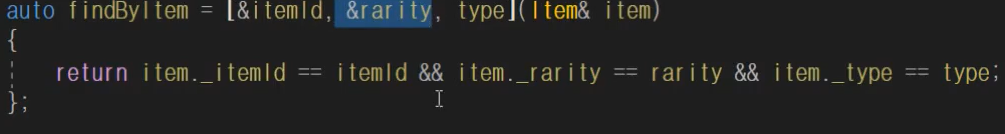
내 포인터를 다른 포인터가 가르킬수없음.

그러나 원본을 이동시킬수 있다는 개념이 있기 때문에 하나만 존재해야하는 포인터이지만

Std::unique\_ptr<Knight> uptr2 = std::move(uptr); 로 원본을 버리고 복사본을 만들수있음.

lambda

* 익명 함수
* [] () {}
* [] : capture라하며 함수 객체 내부에 변수를 저장하는 개념과 유사
* 값 복사 방식 [=], 참조 방식 [&] 람다 밖에 있는 변수 사용할 때 사용
* () : 함수의 매개변수
* {} : 함수 내용
* 람다에 의해 만들어진 실행시점 객체를 클로져(closure) 라고 한다.



ex) 람다 밖에있는 변수를 가져올때 []대괄호 안에 복사 or 참조를 선택해 사진처럼 사용

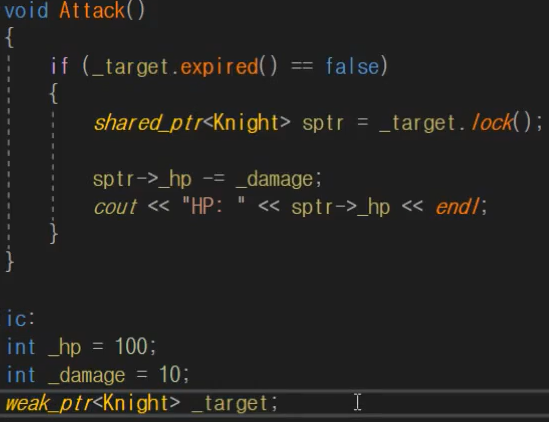
스마트 포인터

shared\_ptr<T>

* 일반 포인터로 만든 객체가 포인터 멤버 변수를 가지고 있고 그 포인터 변수에 어떤 객체를 담는데 그 객체가 소멸될 때 이 객체를 멤버 변수로 가지고 있던 객체는 아직 소멸된 객체의 주소를 가지고 있다. 그 문제를 해결하기 위해 shared\_ptr을 쓰는데 이 포인터로 선언된 객체는 다른 객체의 주소를 멤버변수로 갖고있는데 그 객체가 소멸될 때 소멸시키지않고 더 이상 참조하지 않을 때 소멸시킨다.
* 이 포인터로 선언된 객체는 더 이상 이 객체를 참조하는 객체가없을 때 소멸된다.

weak\_ptr<T>

* shared\_ptr의 경우 서로다른 shared\_ptr 객체가 서로의 주소를 참조하고있을 때 반복 사이클이 돌며 메모리가 삭제되지 않아 언젠가 메모리 오버플로우가 발생한다. 그 사이클을 없애주기위해 weak\_ptr을 쓴다.



unique\_ptr<T>

중요한데 자꾸 까먹는 것

Over loading

* 함수 이름의 재사용, 이름은 같지만 인자가 다른 함수

Override

* 상속 관계에서의 부모 함수 재정의

Const의 위치에 따른 의미

* 상수의 의미 ex) const int num = 10;
* 멤버 함수 오른쪽에 const를 붙여 readonly로만 사용하겠다. 멤버변수를 수정하지못한다.

궁금한 것

Inline 함수

왜 변수들을 하나씩 선언할때는 주소값이 32떨어져서 선언이 되는데

배열로 선언하면 바로 다음 주소에 값들이 들어가는지

클래스안에서의 static 멤버변수

Static 변수 벡터 클래스, 리스트 클래스 만든것처럼 자료구조도 상세하게 공부해서 스택, 힙, 큐, 트리, 등등의 자료구조 함수들을 직접 구현해보기 템플릿사용

언리얼 블루프린트에서 이벤트와 함수의 차이점이 혹시 이벤트는 멀티쓰레드처럼 동시에 작동하지만 함수는