코딩테스트

- 알고리즘 4문제, 서술형 2문제 나옴
- 알고리즘 2문제는 쉬웠고, 컴파일 해볼 수 있음
- 알고리즘 1문제는 merge sort 비슷한 느낌 문제였는데, 의사코드로 손코딩
- 알고리즘 나머지 1문제는 시간 없어서 못품
- 서술형 1문제는 길드 클래스 사양서 주고, 클래스 구현해보라고 함
- 서술형 1문제는 어떤 클래스 코드 주어지고,
    - 1. 이 때의 문제점과 어떻게 개선할 수 있는지를 쓰시오
    - 2. 이 코드가 멀티스레딩 환경에서 왜 문제가 되는지와 어떻게 개선살 수 있는지를 쓰시오

1차 기술면접

- 들어가자마자 문제 8개 풀었음
    - 1. yaw, pitch, roll 방향 표시
    - c++에서 & 연산자 계산
    - 벡터 거리/속도 계산
    - 벡터 내적 이용해서 각도 계산
    - 언리얼에서 오일러 회전 안쓰고 쿼터니언 쓰는 이유 기술
    - radian 뜻
    - DFS, BFS 손코딩
    - 문자열 뒤집기 손코딩
- 기타 질문
    - int 포인터에 1 더하면 어떤 일이 벌어지나요?
    - cast 4가지 종류 설명해보세요
        - dynamic cast 설명해보세요
            - dynamic cast에서 가상함수 테이블이 어떻게 동작하는지
            - (추상클래스의 가상함수 테이블과 구분해서 설명해야 함)
    - 스마트포인터 4가지 종류 설명해보세요
    - 언리얼의 GC에 대해 설명해보세요
- 기타 포폴에 대한 질문들..
    - 이러이렇게 바꾼다면 어떻게 할 수 있을지 방법을 설명해보세요 등등 (잘 기억 안남)

1. **락에 대해 아는대로 설명해보시오 (락, 뮤텍스, 데드락, 세마포어 등...)**

- 락
    - 여러 스레드가 동시에 자원을 접근하는 상황에서 데이터를 보호하기 위한 방법
    - 락을 걸어놓고 들어간 해당 스레드를 제외한 어떤 스레드도 락이 걸린 구역으로 들어가지 못하는 것을 의미
    - 락을 걸어놓은 해당 스레드가 락을 릴리즈(해제)하고 해당 구역에서 나가면, 다른 스레드가 접근 가능
    - 데드락 설명에서는 자원 == 코드구역 이라고 봐도 된다
        - 크리티컬 섹션 안의 명령어들도 스레드가 독점해야 할 자원으로 취급하기 때문

        - 그 해당 스레드에서 다른 스레드로 context switching이 될 수는 있지만, lock이 된 구역은 해당 스레드 외엔 누구도 진입하지 못하게 됨을 의미
    - 메모리를 공유하여 읽기/쓰기를 수행할 스레드들은 공유 메모리에 접근하기 전에 메모리에 대해 락을 시도해야 한다.
    - 읽는 중에 데이터가 바뀌거나 하는 사태를 방지하기 위함
    - 다른 스레드가 이미 점유중일 경우, 새로 락을 시도하려던 스레드는 점유중인 스레드가 락을 해제하거나 타임아웃이 될 때 까지 대기하게 된다.
- 락을 구현하는 방법
    - 세마포어는 화장실 여러개를 세마포어가 관리하는거
    - 뮤텍스는 화장실이 1개이고 들어갈 수 있는 키를 이용해서 사용하는거
    - 뮤텍스, 세마포어 : 동기화 기법
        - 데드락 발생 가능
- 뮤텍스
    - std::mutex
        - 읽기, 쓰기 둘다 하나만 접근
    - std::shared_mutex
        - 읽기는 공유, 쓰기는 하나만 접근
    - 코딩테스트 문제
        - map 대신에 mutex를 쓰는게 아니고, map을 사용함에 있어서 mutex를 이용해 락을 거는 작업이 필요함
            - 직접 사용해본적은 없어서 익숙하지 않음
        - 멀티쓰레드를 고려한 동시성 컨테이너 (concurrent_map)을 사용하는것도 좋음
            - 동기화를 자체적으로 처리해서 좋음
    - 하나의 스레드만 특정 공유 자원에 접근할 수 있도록 보장하는 락의 한 종류
    - 한번에 하나의 스레드만 자원을 사용할 수 있도록 함
    - 사용 중인 스레드가 뮤텍스 해제(unlock)을 할 때 까지 기다려야 함
    - 뮤텍스는 소유권이 있음
        - 자원을 잠근 스레드만이 그 자원 해제 가능
    - std::mutex 같은 라이브러리 사용
        - std::mutex mtx;
        - mtx.lock();
        - mtx.unlock();
    - 뮤텍스의 데드락
        - 다중 락
            - 한 스레드가 뮤텍스 1을 잠그고 뮤텍스 2를 기다리는 중
            - 다른 스레드가 뮤텍스 2를 잠그고 뮤텍스 1을 기다리는 중
- 세마포어
    - 여러 스레드가 제한된 수만큼 자원에 접근할 수 있도록 함
    - 카운터 값을 기준으로 동작
        - 카운터 값이 0이 되면 자원이 모두 사용중이라는 뜻
    - 뮤텍스는 하나의 스레드만 자원에 접근할 수 있는 반면, 세마포어는 여러 스레드가 접근 가능하면서 접근 가능한 수를 조정
    - std::counting_semaphore<3> sem(3);
        - sem.acquire(); // 카운터 감소
        - sem.release(); // 카운터 증가
    - 상호 배제 알고리즘 (Mutual Exclusion Algorithm)에 기반

    - 동시 프로그래밍에서 공유 불가능한 자원의 동시 사용을 피하기 위해 사용되는 알고리즘
    - 임계 구역에서 구현된다
- 원자적(Atomic) 으로 제어되는 정수 변수
- 일반적으로 세마포어의 값이 0이면, 자원에 접근할 수 없도록 블럭(Block)을 하고 0보다 크면 접근함과 동시에 세마포어의 값을 1 감소시킨다.
- 종료하고 나갈 때는 세마포어의 값을 1 증가시켜 다른 프로세스가 접근할 수 있도록 한다
    - 여기서 접근되는 자원은 임계 구역 (Critical Section)으로, 이 설정에 따라서 프로그램의 퍼포먼스가 극단적으로 하락할 수 있어 주의 필요
    - 임계 구역/영역 (Critical Section)
        - 여러 프로세스가 데이터를 공유하며 수행될 때, 각 프로세스에서 공유 데이터를 접근할 수 있도록 하는 프로그램 코드 부분
        - 즉 공유되는 프로그램 코드 부분
        - 여러 프로세스가 동일 자원을 동시에 참조하여 값이 오염될 위험 가능성이 있는 영역
        - 프로그래밍 시 임계 영역을 최소화 하는 설계를 해야 한다.
- 세마포어도 데드락 발생 가능
- 구성
    - 변수 S
        - 일반적으로 정수형 변수
        - 세마포어의 종류
            - 이진형 세마포어
                - 0 또는 1을 가짐
                - 즉, 1개의 공유 자원을 상호 배제하며 이를 이용해 계수 세마포어를 구현할 수도 있다.
            - 계수형 세마포어
                - 0과 양의 정수값을 가질 수 있다
                - 여러개의 공유 자원을 상호배제할 수 있다
        - P와 V라는 명령에 의해서만 접근 가능
    - P 연산
        - try를 뜻하는 네덜란드어
        - 임계 영역을 사용하려는 프로세스들의 진입 여부를 결정하는 조작
        - Wait 동작이라고도 함
    - V 연산
        - increment를 뜻하는 네덜란드어
        - 대기중인 프로세스를 깨우는 신호 (wake-up)
        - signal 동작이라고도 함
    - P는 임계 구역에 들어가기 전에 수행되고, V는 임계구역에서 나올 때 수행된다
    - 변수의 값을 수정하는 연산은 모두 원자성을 만족해야 한다.
        - 즉, 한 프로세스(또는 스레드)에서 세마포어 값을 변경하는 동안 다른 프로세스가 동시에 이 값을 변경해서는 안된다.
    - 두 개의 atomic operation인 wait(P)와 signal(V)로만 접근이 가능한 방법이다.
    - P = 세마포어 카운트를 1 깎고 연산에 진입

            - 이미 자원을 사용하는 task가 있다면 재움 큐에서 자고 있게 된다.
        - V = 세마포어 카운트를 1 추가하고 연산에서 빠져나온다.
            - 대기하고 있는 task가 있다면 재움 큐에서 자고 있는 연산을 깨워준다. - busy waiting 방지
        - 세마포어는 0 이하로 떨어질 수 없다.
        - P와 V가 atomic 하기 때문에, P->V, V->P 등의 동작을 하게 될 경우 데드락이 발생하거나, 상호배제(각 스레드가 자원에 대해 서로를 배타적으로 배제함)를 보장할 수 없게 되는 약점 있다
            - 해결 방법으로는 세마포어가 모든 자원을 다 얻은 채로 프로세스를 진행하게 하거나(다 얻지 못하면 그 즉시 모든 자원을 반납하고 재시도한다), 타임아웃을 넣거나 하는 방법이 있다
- 데드락
    - 두 개 이상의 스레드가 서로 자원을 기다리며 교착 상태에 빠지는 현상
    - 해결 방법
        - 락 순서를 정하거나, 타임아웃 설정
    - A, B 자원을 둘다 사용해야 하는 스레드 C, D 가 있는데
        - 스레드 C가 A를, 스레드 D가 B를 가지고 서로의 객체를 반납하기만을 기다리고 있는 상태
    - 특정한 자원(메모리, 레지스터, CPU 연산장치 등을 의미)을 사용하는 스레드들이 서로의 자원을 반납하기만을 무한히 기다리는 현상
    - 멀티 스레드 환경에서는 매우 잘 발생 가능, 발생하면 해결하기 어려움
    - 두 개 이상의 스레드가 서로 상대방이 점유한 자원을 기다리면서 영원히 대기 상태에 빠지는 현상
    - 락 순서를 정하거나, 타임아웃을 설정하는 방법으로 해결할 수 있다
    - 자원 점유 전에 모든 자원을 한꺼번에 확보하는 방식도 사용됨

16. **동기화 구조체에 대해 설명해보세요**
- 동기화 구조체
    - 멀티 스레드 환경에서 스레드 간 동시성을 제어하는데 사용되는 것
    - 주로 자원을 안전하게 보호하거나 스레드 간의 작업을 조율하는 역할
- 대표적으로 뮤텍스, 세마포어, 조건 변수
    - 뮤텍스
        - 상호 배제를 보장하는 락
        - 하나의 스레드만 자원에 접근하고 다른 스레드는 뮤텍스가 해제될 때까지 기다리게 됨
    - 세마포어
        - 여러 스레드가 자원에 접근할 수 있는 횟수를 제한하는 구조체
        - 한정된 자원에 대해 동시에 여러 스레드가 접근할 수 있을 때 유용
    - 조건 변수
        - 스레드가 특정 조건이 충족될 때 까지 기다리도록 하고, 조건이 만족되면 다른 스레드에게 신호를 보내 작업을 재개하도록 하는 방식으로 동기화

3. **스마트 포인터에 대해 잘 알고 있는가**
- 참조자 (Reference)

    - 실체가 있어야 하며(별명을 붙일 놈이 필요함) 선언 즉시 할당되어야 함
    - 즉 NULL, nullptr로 할당 불가능
    - 레퍼런스는 초기화리스트를 사용하여 먼저 초기화 해야함 (modern c++의 초기화리스트와 다름, 생성자의 초기화리스트를 의미함)
        - 초기화 리스트
            - 생성자 본체가 실행되기 전에 멤버 변수를 초기화 하는 방식
            - 콜론 (:) 사용
            - 참고로 초기화 리스트는 상수, 레퍼런스, has-a관계의(포함한) 클래스 초기화에 사용해야 함
    - 생성자 내부에서의 초기화는 먼저 null로 생성한 뒤 값을 넣는 방식이기 때문
    - 또한 한번 할당하면 다른 곳에 재할당 불가능
        - 포인터는 다른 메모리 주소로 재할당 가능
        - 즉 가리키는 주소 변경 가능
    - 레퍼런스는 참조하는 대상이 고정 (변경 불가)

- 포인터(Pointer) 타입 변수
    - 주소값을 저장할 수 있는 타입의 변수
    - 실체가 없이 null 가능, 언제든 할당할 수 있음
    - 또한 동적 메모리 할당에 사용함
    - 포인터는 초기화리스트 사용하지 않고도 동적할당 또는 주소 대입을 통해 초기화 가능

- Malloc : 단순한 메모리 할당. 할당 시 메모리의 사이즈를 입력해서 할당받음. C스타일
    - void*를 리턴하기 때문에 원하는 타입으로 캐스팅해서 사용
    - int* c_style = (int*)malloc(sizeof(int) * 10);
- New
    - 할당과 동시에 초기화 가능(초기값을 줄 수 있음)
    - 생성자 호출됨 (즉 C++ 객체 할당에 사용)
    - 오버로딩 가능 (new도 연산자이다)
    - 할당 시 객체의 크기를 입력하여 할당 받음. C++ 스타일
        - int* cpp_style = new int[10];

- 댕글링 포인터
    - 이미 해제된 메모리나 유효하지 않은 메모리 주소를 가리키는 포인터
        - 어떤 주소를 가리키고 있지만,
        - 그 주소에 있는 메모리는 더이상 유효하지 않을 때
    - 발생 상황
        - 메모리 해제 후 포인터 유지될 때
        - 스택 메모리 범위 벗어난 경우
            - 함수 지역 변수 참조하는 경우, 함수 종료되면 지역변수 해제
        - 객체 소멸된 후에 포인터가 남아있을 때
    - 방지 방법
        - 메모리 해제 후 nullptr로 설정
        - 스마트 포인터 사용
        - 지역 변수의 주소 반환하지 않도록 주의

        - 동적 할당을 통해 변수를 할당하고, 그 주소를 반환하는 것이 안전
    - 객체의 수명 관리 주의
- 포인터 변수를 delete나 free 할 시에 메모리가 할당, 해제되었다 해도, 변수가 가리키는 주소값이 사라지는 것이 아니기 때문에, 그 포인터 변수를 다시 참조하려고 하면 미정의 동작을 수행한다.
- 따라서, 메모리를 해제하는 구문 이후, 해당 포인터 변수를 nullptr로 바꿔주고 사용할 때마다 nullptr인지 체크하는 테크닉 필요
- 혹은 스마트 포인터 쓰면 메모리 누수, 버그 어느정도 해소 가능

- 스마트 포인터
    - 원래 C++에서 동적 메모리를 할당하기 위해서는 new / delete를 이용
    - 이러한 포인터를 원시 포인터(날 포인터, 쌩 포인터, raw pointer)라고 함
    - 프로그래머가 책임지고 delete를 통해 사용되지 않는 메모리를 반환해야 했음
    - 메모리를 반환하지 않고 메모리 누수가 발생하거나 적절하지 못한 타이밍에 메모리를 delete하여 댕글링 포인터를 발생시켜 프로그램의 안전을 위협할 수 있음, 특히 찾기 어려움
    - 스마트 포인터는 C++11에 공식적으로 추가된 기능
        - 객체가 더는 필요하지 않을 때 객체에 할당된 메모리가 자동으로 해제
        - 즉 메모리 누수의 가능성을 제거
        - 레퍼런스 카운트를 통하여 제거 시점을 결정
        - <memory> 헤더 안에 정의되어 있음
        - 원시 포인터와 마찬가지로 역참조를 통해 객체에 접근 가능
        - 임의의 크기로 스마트 포인터 배열도 생성 가능
    - unique_ptr<T>
        - 언제나 유일해야 함
        - 즉 둘 이상의 unique_ptr이 같은 주소를 가질 수 없으며, 복사될 수도 없음
        - 유일성을 손상시켜서는 안된다
        - unique_ptr을 옮기려면 오직 소유권을 이동시키는 행위만이 허용
        - <utility> 헤더의 std::move() 사용하면 unique_ptr 객체에 저장된 주소를 다른 unique_ptr로 이동시킬 수 있으며, 소유권을 이전하면 기존의 unique_ptr 객체는 무효화 된다.
        - 객체에 대한 단일 소유권만 허용하고 싶을 때 사용하는 것이 좋다.
        - ex)
            - unique_ptr<string> P { new string { “AY” } };
            - auto P = make_unique<string> (“AY”);
            - auto P = make_unique<string> (6, ‘A’);
        - 복제가 불가능 하므로, 함수에 값으로 전달할 수 없다.
        - 만약 함수의 인수로 쓰고 싶다면 참조 매개변수로 받아야 한다
        - 복제될 수는 없지만 암묵적 이동 연산 (implicit move operation)에 의해 반환 가능하므로 함수에서 반환될 수 있다.
        - reset()
            - 스마트 포인터가 가리키는 원본 객체를 소멸, unique_ptr은 해제된다.
            - ptr.reset(new string{“AY”})와 같이 reset의 인자에 새 객체를 넣으면 스마트포인터에 새 객체를 연결해준다.

- release()
    - 스마트 포인터가 가리키는 원본 객체의 소유권을 해제하고, 원본 객체를 리턴
- unique_ptr 객체끼리 비교하고 싶을 때는 두 객체의 get()을 호출하여 반환된 주소값을 비교

- shared_ptr<T>
    - unique_ptr과 반대로 객체를 여러 shared_ptr과 공유 가능
    - 레퍼런스 카운팅 방식을 사용하여 메모리 관리
        - 새로운 shared_ptr 객체가 주소를 공유받을 때마다 레퍼런스 카운트가 증가
        - 공유를 받았던 shared_ptr 객체가 소멸되거나, 다른 주소를 할당받거나, nullptr을 할당받으면 레퍼런스 카운트 감소
        - 만약 레퍼런스 카운트가 0이 되면 해당 주소를 위한 힙 메모리가 자동으로 해제
            - 힙 영역
                - 동적으로 할당되는 메모리 영역
    - 새로운 shared_ptr을 정의하면 두가지 할당을 받음
        - shared_ptr이 가리키는 원본 객체를 위한 힙 메모리를 할당
        - 스마트 포인터의 레퍼런스 카운트를 위한 컨트롤 블록을 위해 스마트 포인터 객체와 관련된 힙 메모리를 할당
    - ex)
        - shared_ptr<double> P = { new double(999.0) };
            - 일반적인 할당 방법
        - shared_ptr<double> P = make_shared<double>(999.0);
            - 더 효율적으로 동작, 추천되는 방법
        - shared_ptr<double> P2 { P };
            - 다른 shared_ptr로 초기화, 레퍼런스 카운트 증가
    - reset() : unique처럼 인수 넣을 수 있음
        - 인수 없이 호출하면 레퍼런스 카운트 1 감소
        - 해당 shared_ptr 객체가 아무것도 가리키지 않게 됨 (unique와 다름)
        - 인수로 원시 포인터 전달하면 shared_ptr이 가리키는 주소 바꿀 수 있다.
    - shared_ptr끼리는 == 연산자를 사용하여 비교 가능
        - 단 두개가 모두 nullptr 일수 있음 주의
        - 스마트 포인터는 암묵적으로 bool로 변형될 수 있으므로, 객체가 있으면 true, 없으면 false
        - if (pA == pB && (pA != nullptr))
        - if (pA == pB && pA)
    - .use_count() : shared_ptr이 가리키는 객체의 레퍼런스 카운트 반환
        - nullptr이라면 0 반환
    - .unique() : shared_ptr이 유일한지(true), 복제본이 있는지(false) 반환

- weak_ptr<T>
    - shared_ptr의 상호 참조 문제 해결
    - shared_ptr 객체에서 생성하여 연결하며, 같은 주소를 가리킨다

        - 단 weak_ptr은 shared_ptr 객체의 레퍼런스 카운트를 증가시키지 않음
            - 객체의 소멸에 관여하지 않음
        - shared_ptr의 메모리가 레퍼런스 카운트가 0이 되어 해제되더라도 연관된 weak_ptr 객체는 남아있게 됨

    - RAII와 동적메모리 자원 관리
        - Resource Acquisition Is Initialization
            - 자원 획득은 초기화 이다
            - 디자인 패턴
        - 자원 획득을 객체 초기화 시에만 하라
        - 자원 획득이 필요한 경우, 자원 획득을 담당하는 클래스를 만들어 그 클래스의 생성자에서만 자원 획득을 해라
        - 자원의 생애주기(자원 획득과 해제)를 객체의 생애에 바인딩 시키는 기법

        - 필요성
            - 자원의 생애주기를 객체의 생애주기에 결합
            - 자원을 사용하고자 하는 상황에서, 생성자에서 자원 획득을, 소멸자에서 자원 해제를 하는 자원 관리용 클래스를 만드는 프로그래밍 패턴
            - 객체의 시작과 끝은 컴퓨터가 알아서 해주는 이점을 활용
            - 객체의 생에는 런타임이 알아서 잘 관리
            - 객체와 바인딩된 자원도 런타임이 자동으로 관리 가능

4. **선형보간과 구면보간의 원리를 알고 있는가**
- 선형 보간 (Linear Interpolation, Lerp)
    - 두 점 사이를 직선으로 연결하여 중간 값을 구하는 방식
- 구면 보간(Slerp)
    - 구면 상의 두 점을 잇는 호를 따라 중간 값을 구하는 방법
    - 회전(쿼터니언) 등에서 주로 사용
    - 일정한 각속도를 유지해 자연스러운 회전 구현 가능

6. **팩토리얼을 다양한 방법으로 구현해보시오(3가지)**
1. **반복문을 이용한 방법**: 단순히 `for`문을 사용해 팩토리얼 값을 곱하는 방식.
2. **재귀함수를 이용한 방법**: `n! = n * (n-1)!` 관계식을 사용해 함수 내에서 재귀적으로 호출.
3. **메모이제이션**: 재귀함수에 캐싱을 적용해 이미 계산된 값을 저장하여 불필요한 재계산을 방지하는 방식.

15. **for문에서 i++(후위 연산자)와 ++i(전위 연산자)의 차이점이 있나요? 속도 차이가 날까요?**
- i++
    - 값이 사용된 후 증가
- ++i
    - 먼저 증가한 후 값을 사용

- 객체 타입에서는 i++가 더 많은 연산을 요구해 미세한 성능 차이가 있을 수 있지만, 일반적인 정수형에서는 거의 차이가 없음

21. **32비트 환경과 64비트 환경은 무슨 차이가 있는가**
- 8비트 = 1바이트
- 32비트 환경
    - 한번에 처리할 수 있는 데이터 크기가 32비트로 제한
    - 메모리 주소 공간이 4GB로 한정
- 64비트 환경
    - 한번에 처리할 수 있는 데이터 크기가 64비트
    - 더 많은 메모리 (이론적으로는 16엑사바이트)를 사용 가능
    - 성능 향상과 보안 기능 개선

27. **메모리 단편화의 두 가지 경우를 설명하고, 해결 방법을 제시하시오**
- 내부 단편화
    - 할당된 메모리보다 작은 데이터가 저장될 때 남는 공간이 발생하는 문제
    - 메모리 할당을 4 단위로 했는데, 만약 7만큼 할당 필요하다면 4짜리 2개를 할당해주어야 하고, 1만큼이 낭비된다
- 외부 단편화
    - 메모리가 해제된 후 인접하지 않은 빈 공간들이 남아 새로운 할당이 어려워지는 문제
    - 전체 메모리에 빈 물리 메모리 공간이 20, 15 있고, 새로 할당해야 할 메모리 크기 연속 21이면, 남은 공간이 충분함에도 연속적이지 않아서 할당할 수 없다.
- 단편화의 해결 방법
    - 메모리 풀
        - 고정된 크기의 블록을 할당하여 malloc이나 C++의 new 연산자와 유사한 메모리 동적 할당을 가능하게 한다
            - malloc이나 new 연산자 같은 기능들은 다양한 블록사이즈 때문에 단편화를 유발시키고, 파편화된 메모리들은 퍼포먼스 때문에 실시간 시스템에서 사용될 수 없게 된다.
        - 동일한 사이즈의 메모리 블록들을 미리 할당해놓는 것
            - 응용 프로그램들은 실행 시간에 핸들에 의해서 표현되는 블록들을 할당하고, 접근하고, 해제할 수 있다
        - 프로그래머가 할 수 있는 외부, 내부 단편화의 해결법이다.
            - 프로그래머가 직접 메모리를 관리할 수 있는 특정 크기의 풀을 만들어서
            - 그 풀에서 메모리를 가져오고
            - 사용이 끝나면 돌려주는 것
            - 미리만든 풀에서 메모리를 가져오므로 할당이 새로 일어나지 않는다.
        - 장점
            - 단편화 완화 가능
            - 할당/해제가 빈번할 때, new/delete 를 통해 새로 할당/해제할 필요가 없이 풀에서 가져오고, 반납하면 되므로 비용이 크게 감소

        - 단점
            - 메모리 풀은 프로그래머가 직접 메모리를 0부터 100까지 관리하겠다고 선언하는 것과 다름이 없다
            - 즉 모든 메모리 문제의 책임을 프로그래머가 가짐
            - 또한 사용하지 않을 때에도 풀을 유지해야 하므로 메모리 낭비가 된다.
    - 페이징
        - 가상 메모리를 페이지로 나누어 다루는 기법
        - 페이지(가상 메모리 나눔의 단위)와 프레임(물리 메모리 나눔의 단위)은 1:1로 대응되며 (가상 메모리 주소값고 물리 메모리의 주소값은 다르므로, 가상 메모리 주소를 기반으로 물리 메모리 주소를 찾아가서 데이터를 가져온다)
        - 이로 인해 물리적으로 연속적이지 않아도 가상 메모리에서 연속된 메모리를 할당할 수 있기 때문에 외부 단편화를 완화
        - 물리적으로 연속적이지 않아도 가상주소는 연속적으로 만들 수 있다
        - 1:1 관계를 '페이지 테이블'에 기록하여, 페이지를 통해 프레임을 찾아서 원하는 데이터를 얻을 수 있게 만든다.
            - 페이지 테이블은 메인 메모리에 저장됨
            - 즉, 가상 메모리 주소를 근거로 페이지 테이블에서 물리 메모리 주소를 찾아서, 물리 메모리 데이터에 접근하는 것 (실제 데이터는 물리 메모리에 들어있으므로)
    - 페이지 폴트
        - 물리 메모리 안에서 페이지에 대응되는 프레임이 실제로 존재하지 않을 때 발생
        - 가상 메모리를 통해 각각의 프로세서들을 전체 메모리를 혼자 점유한다고 착각
            - 실제 메모리는 이런 착각을 지원하기 위해 물리 메모리가 가득 차면 사용하지 않을 것 같은 데이터를 메모리에서 내쫓고 새로운 데이터를 들여오기 때문에 필요한 데이터가 없을 수 있다
        - 디스크를 이용한 데이터 로드가 필요하기 때문에 성능적 비용 매우 큼
        - 최대한 피해야함
        - 쉽게 말해 메모리가 부족하면 생기는 현상
    - 디맨드 페이징
        - 페이지 폴트가 발생하면, 결국 디스크에서 데이터를 가져와야 함
        - 이를 요구 페이징이라고 하며 매우 느린 저장장치인 2차 저장장치(하드디스크)에서 가져오는 것이라 많은 시간 손실 발생
-

30. **객체지향과 절차지향의 차이점을 설명하고, 왜 객체지향을 쓰는지를 설명하시오**

- C++는 멀티 패러다임 언어
    - 객체 지향, 절차 지향 둘 다 지원
- 절차 지향
    - 프로그램을 순차적인 절차(함수)로 나누어 문제 해결하는 방식
    - 함수들의 순서에 초점
    - 흐름에 집중
    - 데이터는 따로 관리

        - 전역 또는 함수 내에서 별도로 처리
    - 코드 재사용 어려움
    - 모듈화 제한적
    - 프로그램 변경 시 수정 많아짐
    - 데이터가 전역적으로 노출될 수 있음
    - C 언어
- 객체 지향
    - 데이터와 그 데이터를 처리하는 함수를 객체라는 단위로 묶어 관리
    - 프로그램을 데이터와 수행할 수 있는 동작(메서드)로 나누어 처리
    - 데이터는 객체 내에서 관리
        - 객체 외부에서 접근 불가
    - 코드 재사용성, 유지보수성, 확장성을 높일 수 있음
        - 상속, 다형성을 통해.
    - 새로운 객체 추가하거나 기존 객체 수정 용이
    - 데이터 은닉을 통해 객체 외부에서 접근 제한 가능
    - 실제 세상을 더 잘 모델링 할 수 있기 때문에 복잡한 프로그램에 적합
    - C++, 자바, 파이썬, C#
- 왜 객체지향을 쓰는가?
    - 모듈화
        - 객체 단위로 코드 구성
        - 유지 보수 수월
        - 각 객체는 독립적으로 개발 및 테스트 가능
    - 재사용성
        - 상속과 다형성을 통해 기존 코드 재사용 가능
        - 중복 코드 줄일 수 있음
        - 확장성 높아짐
    - 유지보수성
        - 최소한의 수정만으로 대응 가능
        - 새로운 기능 추가, 기존 기능 변경할 때에도 영향 적음
    - 현실 세계 모델링
        - 현실 세계의 사물과 개념을 그대로 객체로 추상화하여 프로그램 구성 가능
        - 직관적인 설계 가능
    - 데이터 은닉성
        - 데이터를 객체 내부에 감추고
        - 메서드를 통해서만 접근하도록 설계 가능
        - 보안성 확보
- 객체지향의 4대 특징
    - 캡슐화 (Encapsualation)
        - 클래스를 통해 변수와 함수를 하나의 단위로 묶는다
        - 같은 역할을 하는 변수, 함수를 모아두었기 때문에 의존성, 커플링이 줄어들어, 관리가 용이해진다
        - 외부에서 직접 접근 못하도록 데이터 보호
        - 객체 내부에서만 데이터 다룰 수 있는 메서드 제공
        - 외부에서는 메서드를 통해서만 데이터에 접근
        - private, public 접근지정자
    - 은닉성 (Information Hiding), 추상화
        - 필요한 기능이나 속성만 보여주고 불필요한 세부사항은 감추는 것

    - 프로그램의 세부 구현을 감추는 것
    - 꼭 필요한 정보만 노출
    - 캡슐화를 통해 이뤄짐
        - 은닉성은 캡슐화를 통해 달성되는 특정한 목적
    - 왜? 의도하지 않은 접근 제한, 잘못쓰는 것 방지 위해
    - 다른 사람이 직접 접근하여 값을 바꾸는 것을 막는다.
    - 코드 사용자는 내부 구현 신경 쓸 필요 없이 메소드를 통해 기능 호출하기만 하면 됨
    - 객체들끼리 서로 구현과 상태를 상관하지 않고 단지 사용하게만 함으로써 의존성을 낮춤
    - 추상 클래스, 인터페이스
- 상속 (Inheritance)
    - 자식 클래스가 부모 클래스의 변수, 함수를 물려받는 것
    - 자식 클래스는 오버라이딩을 통해 자신만을 특성 디테일하게 구현
    - 특정 기능 재사용 가능
    - 코드 재사용성, 유지보수성 향상
    - 왜?
        - 다형성을 위해
        - 부모와 자식을 다형성으로 엮어놓으면, 부모에서 추상적인 형태를 주고, 자식에서 세밀화하여 유연하게 코드 작성 가능
    - 추상화 : 공통의 속성이나 기능을 묶어 이름을 붙이는 것
- 다형성 (Polymorphism)
    - 여러가지 형태를 가질 수 있는 능력
    - 동일한 인터페이스나 메서드가 여러 형태로 동작할 수 있는 성질
    - 실제 객체 타입에 따라 다르게 동작 가능
    - 컴파일 타임 다형성(정적 다형성)
        - 함수 오버로딩, 연산자 오버로딩
    - 런타임 다형성(동적 다형성)
        - 가상함수와 동적 바인딩을 통해 구현
        - 추상클래스
        - 상속
    - 1) 상속에 의한 다형성
    - 2) 오버로딩
        - 같은 이름의 함수라도 인자(parameter)타입이 다르거나 개수가 다르면 다른 함수로 정의해서 쓸 수 있음 (리턴 타입은 오버로딩과 아무 관련 없음, 리턴 타입으로 함수를 구분할 수 없기 때문)
    - 3) 오버라이딩
        - 상속 관계에서 부모의 메서드를 자식이 자기에 맞게 재정의해서 사용
    - 4) 템플릿에 의한 다형성
        - 클래스, 함수를 타입에 독립적이게 만드는 도구
        - 여러 타입에 대응되는 단 하나의 객체나 함수 만들 수 있음
        - 함수와 클래스를 개별적으로 다시 작성하지 않고도 각기 다른 수많은 자료형에서 동작 가능
        - 템플릿의 타입은 컴파일 타임에 결정되어 인스턴스화
            - 컴파일 타임에 컴파일러가 실제 코드 구현부를 모두 볼 수 있어야 함

        - 인스턴스화
            - 컴파일러가 컴파일 시 요구되는 타입으로 클래스 정의 코드를 생성해내는 것, 즉, 타입을 지정하여 그 타입을 사용하는 클래스 만들어 내는 것
        - ex) 만약 float 타입으로 템플릿 클래스 사용 -> 컴파일 타임에 컴파일러가 float 타입의 클래스를 만들어낸다. 이를 인스턴스화 라고 함
        - 그렇기 때문에 템플릿 클래스의 함수 구현은 헤더파일에 들어가야 함
        - 혹은 헤더파일에서, 템플릿 정의 부분 아래에 cpp 파일을 인클루드 하는 방법도 있음 -> 이 때는 cpp파일을 빌드 리스트에 넣지 말아야 함

33. **C++의 4가지 캐스팅에 대해 설명해보시오**

- const_cast
    - 해당 포인터 객체의 상수성을 제거/추가
    - const 또는 volatile 속성을 추가하거나 제거할 때 사용
    - 보통 상수성은 항상 유지되어야 하지만, 외부의 다른 라이브러리르 사용할 때 인자값으로 비상수성 객체를 요구한다던지 하는 불가피한 상황에서 한시적으로 사용
    - 단 const_cast를 사용하는 것은 해당 객체를 인자로 받는 함수 내에서 해당 상수 변수를 수정하지 않는 다는 것을 알고 있을 때만 사용해야 한다.
    - 만약 함수 내에서 객체의 내용이 바뀐다면, 상수 제한을 잠깐 푸는 것에 그치지 않고 상수성 자체를 잃어버리는 것이기 때문에 문제가 생기게 된다.
    - 이럴 땐 유도리 있게 사용할 것이 아니라, 프로그램의 전체 구조를 다시 생각해보아야 한다.
- static_cast
    - 일반적인 캐스팅 방식
    - 컴파일 타임에 타입 변환을 수행
    - 단점 : 런타임 타입 검사를 하지 않음, 안전성이 보장되지 않음
    - 업캐스팅, 다운캐스팅 둘 다 가능
        - 계층 구조가 실제로 유효한지는 검사하지 않음
        - 즉, 자식 클래스를 부모 클래스로 업캐스팅 하여 사용하다가(다형성을 위해)
        - 다시 자식 클래스로 다운캐스팅하는 것은 유효하며, 정상적인 사용방법이지만,
        - 순수한 부모 클래스인 객체를 자식으로 다운캐스팅하여 사용하는 것도 막지 않는다는 문제
            - 메모리 침범 등 심각한 오류 발생
    - 계층 구조 간 안전한 캐스팅을 하려면 dynamic_cast를 사용해야 한다.
    - 컴파일 타임에 캐스팅을 수행하지만, 타입 안정성이 보장되지 않아 잘못된 캐스팅 시 예상치 못한 동작 일으킬 수 있음
- dynamic_cast
    - virtual 함수가 하나라도 있어야 함.
        - vtable에 RTTI가 저장되기 때문에.
        - RTTI : RunTime Type Information
    - 런타임 타입 검사를 수행하는 안전한 다운캐스팅에 사용하는 방법
        - 실패 시 nullptr 반환

        - 다이나믹 캐스트는 타입이 유효한지 체크하기 위해 런타임에 타입 검사를 수행하기 때문에, 런타임 비용이 높아서 잘 사용되지 않는다.
        - 런타임에 안전하게 다운캐스팅
        - 실패할 경우 nullptr 반환하여 캐스팅 오류를 방지
    - reinterptret_cast
        - 타입을 강제 변환. (메모리 재해석)
        - 매우 위험할 수 있음
        - double을 byte로 변환 하는 등 임의 변환 가능
        - 타입을 비트 단위로 1:1 변환
        - 그렇기에 원본 데이터가 소실되거나(데이터가 잘리거나), 원하지 않은 결과가 나올 수 있다

32. **정렬(sort), GC, Thread에 대해 설명하시오**
- 정렬(sort)
    - 데이터를 순서대로 나열하는 알고리즘
    - 퀵소트, 머지소트 등 각각 특징, 방법, 시간복잡도

40. **자료구조가 뭔가요**
- 자료구조
    - 데이터를 효율적으로 저장하고 관리하기 위한 구조
    - 배열, 리스트, 트리, 맵 등
    - 삽입, 삭제, 탐색, 정렬 등 작업을 최적화 하기 위해 사용됨
    - 각각 자료구조는 특정 상황에서 효율성을 극대화할 수 있도록 설계되어 있음

41. **배열과 리스트 차이점**
- 배열
    - 고정 크기의 메모리 블록을 사용해 연속된 메모리 공간에 데이터 저장하는 자료구조
    - 동적 크기 변경 불가능
    - 메모리 상에 연속적으로 저장되는 데이터 구조
    - 인덱스를 통한 빠른 접근 가능 O(1)
    - 삽입, 삭제 항상 O(n) (모든 요소 이동)
- 벡터 (Vector)
    - 동적 배열(크기 조절 가능)
    - 임의 접근 지원
    - 자동 메모리 관리
        - 더 많은 데이터 추가되면 자동으로 메모리 크기 증가
        - 기존 요소는 새로운 메모리로 복사
    - 단점
        - 추가, 삭제 비용
            - 끝에 추가 할 때는 빠르지만
            - 중간에 삽입, 삭제하면 모든 요소 시프트 -> O(n)
        - 크기 늘어날 때 메모리 재할당으로 오버헤드 가능성
    - 배열과 같이 연속된 자료 구조 이므로, 캐시 친화적.
    - 임의접근 반복자를 사용하므로 배열의 원소에 즉시 접근 가능
    - (반복자 5가지)
    - emplace를 하면 복제 과정 없이 바로 원소를 삽입 가능
        - push_back보다 비용 절감

        - 반복자 무효화가 발생 가능
            - 반복자 무효화
                - 컨테이너의 메모리가 재할당 될 때(resize, push_back 등) 또는 요소 삭제를 하는 등의 동작에서 발생 가능
                - 만약 STL의 erase를 사용한다면, erase한 원소를 포함해서 뒤의 원소를 가리키는 모든 반복자가 무효화. 이는 삭제 후 뒤에 있는 요소를 모두 당겨줘야 하기 때문.
                - (erase 함수는 다음 주소를 가리키는 iterator를 리턴)
        - 벡터 요소의 삽입
            - 삽입한 위치 뒤를 가리키는 반복자들은 무효화 된다.
                - 나머지 원소를 모두 밀어줘야 하기 때문에.
            - 삽입 앞은 재할당에 따라 무효화일수도 있고, 아닐수도 있으나 안전한 프로그래밍을 위해서는 무효화라고 보는 것이 맞다.
                - 위치 변동은 없으나 메모리 재할당(크기 초과로 인한 재할당 등)이 일어날 수 있기 때문
        - 벡터 요소의 삭제
            - 삭제 뒤는 무효화(뒤에 있는 원소들을 당겨줘야 해서)
            - 삭제 앞은 무효화 되지 않음 (재할당이 일어날 일이 없다)

- 리스트 (List)
    - 연결 리스트를 구현한 컨테이너
    - 각 요소가 노드로 저장
        - 노드 끼리는 포인터로 연결
    - std::list는 이중 연결 리스트
    - 동적 크기
    - 빠른 삽입 및 삭제
        - 노드를 연결하는 방식이기 때문에, 임의의 위치에서 삽입, 삭제가 O(1) 시간 안에 가능
        - 특히 중간에 삽입, 삭제 시 매우 효율적
    - 순차 접근만 가능
        - 탐색에는 O(n)
    - 임의 접근 불가
    - 각 노드가 포인터를 저장하기 때문에 배열이나 벡터보다 메모리 사용량이 많음
    - 단/양방향 반복자 사용
        - 임의 접근 불가능
        - 어떤 원소에 접근하기 위해서는 해당 위치까지 순회해야 함
    - 리스트의 advance 함수는 임의 접근처럼 보이지만 실 구현은 순회로 되어 있다.
    - 삽입과 삭제가 상수 시간을 가진다
    - 포인터를 사용하여 서로를 연결
    - 삽입과 삭제가 빈번한 구조에 사용하기 좋다

- 배열은 소량 데이터 고정된 크기
- 벡터는 크기 변경 빈번한 경우
- 리스트는 삽입, 삭제 빈번한 경우

42. **오브젝트 풀(풀링)이 뭔가요**

- 오브젝트 풀
    - 성능을 최적화하기 위해, 자주 사용되는 객체를 미리 생성해두고 필요할 때 재사용하는 기법
    - 메모리 할당과 해제의 오버헤드 줄일 수 있음
    - 주로 리소스가 빈번하게 생성되고 소멸되는 곳에 유용하게 사용
        - 포폴 (공격 날리는거 다시 확인하기)

47. **추상 클래스 설명해보시오**

- 구조체
    - 기본 접근지정자가 public
    - 주로 데이터를 저장하는 용도로 사용
- 클래스
    - 의미 : 사용자 정의 자료형
    - 객체의 상태/특성을 정의하는 일종의 설계도
    - 객체화(인스턴스화) 했을 때 객체가 된다
        - 클래스를 인스턴스화 해야 인스턴스가 됨
        - 인스턴스
            - 클래스를 기반으로 만들어져 실제로 메모리에 할당된 것
    - 변수, 메서드를 가짐
    - 기본 접근지정자가 private

- 구조체/클래스에서 바이트 패딩
    - 패딩
        - 구조체나 클래스를 실제 메모리에 올릴 때, 성능 향상을 위해 추가적인 메모리를 할당하여 끼워 넣는 것
        - 실제 어떻게 하는지 예시
        - 하는 이유
- 추상 클래스
    - 하나 이상의 순수 가상함수를 포함하는 클래스 (= 0)
    - 이를 상속받는 클래스는 반드시 해당 가상함수를 구현해야 함
    - 공통된 인터페이스를 제공하는 것, 구체적 구현은 파생 클래스에서 각각 다르게.
    - 보다 구체적인 클래스가 파생될 수 있는 일반 개념식 역할
    - 추상 클래스는 인스턴스화 할 수 없음
- 인터페이스
    - 클래스가 특정 기능을 구현하도록 강제하는 역할을 함
    - C++에서는
        - 추상 클래스
- 가상함수
    - 부모 클래스에서 정의된 함수
    - 파생 클래스에서 이를 재정의(오버라이딩) 할 수 있도록 해줌
    - 가상함수는 다형성을 지원
        - 상속받은 클래스에서 재정의되어 동적 바인딩이 이루어질 수 있음
        - 동적 바인딩이란
            - 런타임 시점에 호출할 함수가 결정되는 방식
            - 객체의 실제 타입에 따라 적절한 함수 호출
            - 가상함수를 통해 가능 (가상함수 테이블 사용)
        - 정적 바인딩

            - 컴파일 시점에 어떤 함수가 호출될지 결정
            - 고정적이어서 성능 더 빠름
            - 일반적인 함수 호출
- 순수가상함수
    - 함수의 선언만 존재하고, 구현은 제공되지 않음
    - 이를 상속받는 파생 클래스는 반드시 해당 함수를 구현해야 합니다.
    - 순수 가상 함수가 하나라도 포함된 클래스는 추상 클래스라고 함
        - 객체를 직접 생성할 수 없음
    - = 0 으로 표현
    - 내용 없이 형식만 정의된 함수
    - 이 형식대로 만들어서 사용하라 라는 의미
    - 자식 클래스는 이를 '반드시' 오버라이드 해야함
    - 순수 가상함수 선언
        - virtual 키워드
            - 이를 상속하는 모든 클래스에서 재정의 가능
            - 파생 클래스의 재정의 멤버로 변경
            - 상속 관계에서 자식이 해당 함수를 오버라이딩 하기 위해 사용
            - 즉, 부모 함수에서 virtual이 없으면 override 할 수 없다.
            - 또한 virtual 키워드를 사용했을 시
                - 런타임에 가상함수 테이블이 생성되어 올바른 함수를 찾아갈 수 있게 만든다
            - 상속 관계에서 부모의 소멸자엔 반드시 virtual을 붙여야 함
                - 소멸 시에 virtual이 없으면, 올바른 자식 객체까지 찾아가지 못하여 부모의 부분만 소멸되고 자식의 부분은 남는 현상 발생
                - 메모리 누수, 오염
            - 절대 생성자, 소멸자에서 가상함수를 호출하면 안됨
                - 자식 객체 생성 시에 가상함수를 호출하게 되면 부모가 먼저 생성될 때, 자식 객체는 아직 초기화 되지 않은 상태이므로
                - 자식 객체는 자신이 부모 클래스 인 것 처럼 동작
                - 즉 부모 클래스의 virtual이 호출되어 의도하지 않은 동작을 하게 됨
                - 또한 미정의 동작을 할 수 있다.
- 가상함수 테이블
    - 다형성 지원
    - 런타임에 동적 바인딩 지원
    - 가상함수 테이블은 클래스마다 존재 (인스턴스 마다가 아님)
    - 각 클래스 내의 가상함수들의 포인터(주소) 목록
    - 각각의 인스턴스들은 해당 클래스의 가상함수 테이블을 가리키는 포인터 변수를 하나씩 가짐 (vptr)
    - 즉, 가상함수가 있는 클래스의 인스턴스 메모리 크기를 구할 땐, 이 포인터 변수의 크기를 더해줘야 함
        - 32비트 환경이면 4바이트, 64비트 환경이면 8바이트
    - 상속 관계의 클래스들은 가상함수 테이블을 사용해 오버라이드 된 함수를 찾아간다
    - 가상함수 테이블은 해당 클래스의 모든 가상함수의 주소값을 제공한다

        - vftable에는 해당 가상함수들의 주소값이 들어있다. 상속 관계가 A->B이면 클래스 A의 vftable에는 A의 가상함수들
            - A::a(void)
        - 클래스 B가 만약 A의 a 함수를 오버라이딩 했다면, B의 vftable에는
            - B::a(void) 이런식으로 내용 작성됨
        - 자식 클래스의 가상함수 테이블에서는 자식 클래스에서 오버라이드 된 함수를 가리킴
    - 런타임에 가상함수 호출 -> 객체의 vptr 주소로 감 = 가상함수 테이블 -> 해당 가상함수 주소 찾아 호출

7. **포물선 공식을 아시나요**
- 수평속도, 수직속도 어쩌고.. 가속도 어쩌고…

8. **상대와 내 캐릭터가 있을 때, 상대가 내 왼쪽에 있는지, 오른쪽에 있는지, 반경 60도 내에 존재하는지 구할 방법을 말해보세요**
- 캐릭터 정면방향 벡터 기준으로 계산 가능
    - 언리얼 함수 뭐였는지 확인
- 두 캐릭터의 위치벡터 함수 (Location)
- 내적 이용해 각도 구할 수 있음
- 외적 통해 왼쪽인지 오른쪽인지 확인 가능

9. **충돌을 구현해보았나요, 어떻게 구현했나요**
- AABB (축 정렬 바운딩 박스) 또는 충돌 구체 (Sphere Collision) 사용
- 자기소개서에 썼던 내용 정리
    - 거리 계산
    - 충돌 감지를 위한 함수 호출해서 물리 엔진에서 처리
        - 언리얼에서는 OnOverlap 함수를 상용해 충돌 시 발생하는 이벤트 처리 가능

10. **가상 메모리에 대해 설명하시오**
- 메모리
    - 흔히 말하는 '메모리'는 대게 주 메모리(main memory) 인 RAM 공간을 의미
        - 물리 메모리
        - 프로그램은 자기가 사용할 데이터를 이 물리 메모리에 미리 가져와서 필요할 때 사용한다
        - 또한 더이상 데이터가 필요하지 않다면, 메모리에서 추방하여 다른 데이터가 들어올 수 있게 만든다
        - 메인 메모리 = 물리 메모리 = 주 메모리 = 램
- 가상 메모리
    - 가상의 메모리 공간
        - 실제로 사용되는 물리 메모리 공간에 1:1 대응시킨다
    - 쓰는 이유
        - 각각 프로세스가 독립적인 가상 메모리 공간을 가지도록 하기 위함
            - 각 프로세스마다 연속적이고 충분히 큰 가상 메모리를 만듦으로써, 실제 물리 메모리가 어떻든 상관 없이 더 큰 메모리 사용할 수 있게 함
            - 프로세스가 자신이 메모리 자원을 점유하고 있다고 착각할 수 있게 만든다

            - ex) 물리 메모리 16기가
                - 가상 메모리를 16기가로 잡아놓으면 물리 메모리가 얼마나 남아있던 간에 각각의 프로세스는 16기가의 공간을 모두 자유롭게 쓸 수 있다
                    - 착각하게 만든다
            - 프로그래머가 물리 메모리의 사용량이나 점유 현황을 신경쓰지 않고도 구현이 가능하게 만드는 이점
            - 프로세스마다 독립적
            - 프로세스가 각자의 데이터를 함부로 접근하거나 조작할 수 없어서 운영체제의 메모리 안전성을 높인다
        - 외부 단편화를 완화하는 효과
    - 물리 메모리와 논리 메모리를 분리해 사용하는 기법
    - 프로그램은 전체 메모리를 사용하는 것처럼 보이지만, 실제로는 필요한 부분만 메모리에 로드되며, 페이지 단위로 나누어 관리됨
        - 이를 통해 메모리 부족 문제 해결 가능
        - 프로세스 간의 메모리 충돌 방지 가능

5. **언리얼의 가비지 컬렉션의 원리를 아는가? C++의 레퍼런스 카운트와의 차이는 무엇인가**

- C++의 메모리 관리
    - C++은 기본적으로 GC 제공 안됨
    - 메모리 할당과 해제는 프로그래머가 직접 수행해야 함 (new, delete 사용)
    - 대신 있는 자동 메모리 관리 방식은 '스마트 포인터'
        - 레퍼런스 카운팅을 통해 객체의 생명 주기 관리
        - 객체의 참조 카운트가 0이 될때 자동으로 메모리 해제
    - 장점
        - 자동, 직관적, 안전한 메모리 관리
        - 멀티스레드 환경에서의 안전성
    - 단점
        - 레퍼런스가 0이 될 때 참조한 객체가 많으면 시간이 오래 걸릴 수 있으며, 멀티 스레드 환경에서 카운팅에 대한 데이터레이스가 일어날 수 있다
            - 데이터 레이스
                - 두개 이상의 스레드가 동시에 동일한 메모리 위치에 접근하여 버그 발생
        - 순환 참조 발생 가능
            - 두 객체가 서로 참조하고 있는 경우
            - 둘 다 카운트가 0이 되지 않아 메모리 해제가 되지 않는 누수 발생
                - weak_ptr 사용
- 언리얼의 가비지 컬렉션
    - UObject 클래스 기반의 GC 사용
    - UObject를 기반으로 한 객체들은 가비지 컬렉션 시스템에 의해 추적 되고 관리됨
    - 마크 앤 스윕 방식

        - 엔진이 주기적으로 가비지 컬렉션을 실행하여, 더 이상 사용되지 않는 객체들을 수집하고 메모리 해제
            - 마크 단계
                - 활성 객체를 찾음
                - 게임 내에서 참조되고 있는 객체들은 활성 상태로 마크됨
                - 참조 관계에 따라 트래버스하여, 다른 객체들에 의해 참조되는 모든 객체를 확인하고 마크.
            - 스윕 단계
                - 마크되지 않은 객체는 더 이상 참조되지 않으므로, 이를 메모리에서 해제함.
                - 이 과정에서 비활성화 된 객체들은 모두 정리되고, 메모리가 반환됨
                - 마크되지 않은 모든 객체들 대상으로 소멸자 호출, 동적 메모리 해제
            - 객체 승격 (Promotion)
                - Young Generation 에서 오래된 세대로 승격될 수 있는 객체들이 있으면, 이 객체들은 가비지 컬렉션이 덜 빈번하게 일어나는 오래된 세대로 이동
    - 언리얼의 GC는 한번에 수행되는 대신, 여러 프레임에 걸쳐 조금씩 나누어 수행하여 성능 저하를 줄임 (Incremental GC)
    - 언리얼의 GC는 객체의 메모리를 즉시 해제하지 않고, 게임의 적절한 시점에서 해제할 수 있도록 연기하는 방식 (Deferred Deletion)
    - C++에서 관리되는 객체(생포인터, TSharedPtr 등)은 언리얼의 가비지 컬렉션에 포함되지 않으므로, 이런 객체는 여전히 수동으로 관리해야 함
    - 장점
        - 프로그래머가 직접 메모리 해제 안해도 됨, 자동 관리로 메모리 누수 방지
        - 복잡한 참조 구조 관리 가능
    - 단점
        - 가비지 컬렉션 실행 시점에 성능 저하 발생 가능성
        - 프레임 드랍이나 일시적 멈춤 현상
        - 정확한 메모리 해제 타이밍을 제어하기 어려움
        - C++ 객체나 스마트 포인터는 수동 관리 필요
- UCLASS 매크로
    - 이 클래스는 UObject 기반
    - 언리얼에서 클래스를 정의할 때 사용되는 매크로
    - 클래스가 UObject를 기반으로 한 언리얼 리플렉션 시스템에 등록됨
    - 언리얼이 해당 클래스의 메타데이터를 관리 하고, 다양한 엔진 기능에서 이 클래스 활용 가능
- UPROPERTY 매크로
    - 클래스 내의 변수(프로퍼티)를 언리얼 엔진의 리플렉션 시스템에 등록하는 데 사용
    - 언리얼이 이 변수를 추적하고 이를 통해 다양한 기능 제공 가능
        - 가비지 컬렉션 시스템에서 관리됨
        - 블루프린트에서 사용 가능
        - 네트워크 상에서 자동으로 복제됨
- 리플렉션 시스템

        - 언리얼이 정보를 런타임에 확인할 수 있게 함
            - 가비지 컬렉션, 네트워크 복제, 블루프린트와의 통합 등에 사용

28. **프로세스와 스레드에 대해 설명하시오**
- 프로세스
    - 운영체제에서 실행되는 독립된 프로그램 단위
    - 각 프로세스는 자신만의 메모리 공간을 가짐
    - 실행되고 있는 각각의 프로그램이 (심지어 같은 프로그램을 여러 개 실행시켰더라도) 모두 프로세스이다.
        - (작업 관리자 - 프로세스 창에서 본 2개의 비주얼 스튜디오..)
        - 똑같은 비주얼 스튜디오지만 각각은 다른 프로세스
    - 각자 고유한 가상 메모리를 가지고 있기 때문에, 실제 물리 메모리 사용량과는 상관 없이 프로세스 자신이 전체 메모리를 전부 가진 것처럼 작동
        - 이러한 착각을 지원하기 위해 가상메모리 할당과 페이징 같은 기능 제공
            - 가상메모리
            - 페이징
    - 다른 프로세스와 통신하려면 스레드보다 비교적 무겁다.
    - 프로세스를 생성 -> 제거하는 일련의 사이클에서
        - OS는 가상 메모리 범위를 잡고, 프로세스를 세팅하고, 제거 시엔 가상 메모리를 다시 풀어주고 하는 과정에서 스레드보다 더 연산 부담을 가진다
    - 컨텍스트 스위칭이 스레드보다 무겁다 (스위칭에 필요한 정보들이 스레드보다 많기 때문)
- 스레드
    - 프로세스 내에서 실행되는 작은 작업 단위
    - 같은 메모리 공간을 공유 (데이터, 힙 영역 + 코드 영역은 읽기전용이라 안전하게 공유되므로 보통 잘 얘기하지 않음)
    - 스레드는 경량 프로세스로, 다중 작업을 처리할 때 오버헤드가 적음
    - 각각의 프로세스의 실행 흐름
    - 하나의 프로세스에서 여러 개의 스레드를 만들 수 있다.
    - 레지스터와 스택 메모리를 제외한, 힙, 데이터 공간 공유
        - 이로 인해 빠른 스레드 간 통신을 할 수 있으며, 컨텍스트 스위칭이 가볍고 빠르다
        - 레지스터
            - CPU에 있는 매우 빠른 임시 저장 공간
            - 연산 수행할 때 주로 씀
            - 크기는 작지만 속도는 램보다 훨씬 빠름
        - 스레드의 컨텍스트 스위칭
            - FCFS (First Come First Served)
            - RR (Round-Robin : 시분할)
            - SJF (Shortest Job First) 등의 기법이 있다.
    - 메모리를 공유하기 때문에 그에 따른 data race, deadlock 문제를 피할 수 없어서
        - Lock 이나 Atomic을 사용하여 잘 회피해야 한다.

        - 스위칭 될 수 있음
            - 시간이 일정 이상 경과하거나(Timeout)
            - IO 인터럽트가 들어오거나
            - Sleep 등의 코드가 실행되거나
            - 스레드 자신이 CPU 점유를 양보하거나(Yield)

29. **컨텍스트 스위칭에 대해 설명하시오**
- 컨텍스트 스위칭
    - 운영체제가 하나의 스레드나 프로세스에서 다른 스레드나 프로세스로 작업을 전환하는 과정
    - 현재 실행중인 작업의 상태를 저장하고, 새로 전환된 작업의 상태를 복원하는 비용 발생
    - 다중 작업 환경에서 효과적인 스케줄링 필요

34. **언리얼에서 액터와 폰의 차이에 대해 설명해보시오**
- 액터
    - 모든 오브젝트의 기본 클래스
        - 모든 게임 오브젝트는 액터를 상속받음
        - 고정된 오브젝트 일 수도 있음
        - 위치, 회전, 스케일 등 속성
        - 다양한 컴포넌트 포함 가능
- 폰
    - 액터의 하위클래스
    - 플레이어나 AI가 조종 가능
    - 주로 플레이어나 NPC처럼 움직일 수 있는 캐릭터에 사용
    - 입력을 받아 움직일 수 있음
    - 애니메이션 없음
- 캐릭터
    - 폰의 하위 클래스
    - 인간형 캐릭터나 애니메이션이 포함된 캐릭터 표현
    - 걷기, 달리기, 점프 등 물리 기반 이동 로직 있음
    - 스켈레탈 메쉬(애니메이션) 기능 내장
- 컴포넌트
    - 액터의 기능을 확장하고 모듈화
    - 액터에게 모양, 물리적 특성, 소리, 입력 처리 등의 다양한 기능 부여
    - 액터에 특정 기능 추가하는 역할
    - ex)
        - 메쉬 컴포넌트
        - 카메라 컴포넌트

36. **const의 4가지 예시를 주고 설명해보시오**
1. **const 변수**: 변경할 수 없는 상수를 정의합니다.
  ```cpp
  const int x = 10;
  ```

2. **const 함수 인자**: 함수 내부에서 매개변수를 수정하지 않도록 보장합니다.
```cpp
void foo(const int x);
```
3. **const 멤버 함수**: 객체의 멤버 변수를 수정하지 않는 함수입니다.
```cpp
void func() const;
```
4. **const 포인터**: 포인터 또는 포인터가 가리키는 값을 수정하지 않도록 보장합니다.
```cpp
const int* ptr;  // 가리키는 값은 변경 불가
int* const ptr2; // 포인터 자체는 변경 불가
```

- 상수 포인터 (const pointer)
    - int* const ptr = &a;
    - 선언과 동시에 초기화 해줘야 함
    - 우측의 값이 상수가 되어 변경할 수 없음
- 상수에 대한 포인터 (pointer to const)
    - const int* const ptr = &a;
    - 주소와 값 모두 변경할 수 없음
    - 읽기전용

37. **반사 벡터 구하기**
Github 참고

39. **싱글톤 패턴이 뭔가요**
- 싱글톤 패턴
    - 클래스의 인스턴스가 오직 하나만 존재하도록 보장하는 디자인 패턴
    - 전역적으로 접근 가능하면서도 하나만 존재하도록
    - 주로 설정 값이나 로그 관리 등 하나의 인스턴스만 필요할 때 사용
    - 게임 매니저
        - 게임 상태, 레벨 관리, 점수, 시간 등
    - 오디오 매니저
    - 로그 매니저
        - 이벤트, 에러 메시지, 디버그 정보 등 기록
    - 입력 처리 시스템
    - 리소스 매니저
        - 텍스처, 모델, 사운드 등
    - 네트워크 매니저
- 디자인패턴이란

44. **Local Space, World Space, View Space는 뭔가요**
- 로컬 좌표계
    - 특정 오브젝트를 기준으로 한 상대 좌표계
    - 오브젝트의 부모 객체가 있을 경우, 부모 기준으로 위치, 회전, 스케일

- 월드 좌표계
    - 전역 좌표계
    - 월드의 원점 기준 좌표
    - x 앞 뒤 / y 좌 우 / z 위 아래
- 컴포넌트 좌표계
    - 특정 컴포넌트(메쉬, 카메라, 콜리전)의 좌표를 정의하는 좌표계
    - 부모 컴포넌트나 액터를 기준으로 함
- 뷰포트 좌표계
    - 언리얼 에디터에서 화면 상의 2d 좌표
    - 화면상에서 클릭이나 드래그 등 마우스 입력 처리 할때 주로 사용
- 스크린 좌표계
    - 게임 화면에서 UI 요소들의 좌표계
- 카메라 좌표계
    - 카메라를 기준으로 한 상대적인 좌표계
    - 1인칭 혹은 3인칭 게임에서 카메라 뷰에 따라 위치 계산할 때 사용
- 절두체 좌표계

###컴파일 타임과 런타임
- 프로그램이 어떻게 처리되고 실행되는지에 대한 시간적 구분
- 컴파일 타임
    - 프로그램 소스 코드를 컴파일러가 기계어 또는 중간 코드로 변환하는 과정이 일어나는 시간
    - 프로그램이 실행되기 전, 코드를 번역하고 오류를 확인하는 단계
    - 즉, 프로그램이 실행되지 않은 상태에서 이루어지는 모든 작업이 컴파일 타임
    - 문법 검사
    - 타입 검사
    - 코드 최적화
    - 링크
        - 여러 모듈이나 라이브러리를 결합하여 최종 실행 파일 생성 과정
- 런타임
    - 프로그램이 실제로 실행되는 시간
    - 사용자가 프로그램을 실행했을 때, 프로그램이 동작하면서 발생하는 모든 일
    - 실제 코드 실행
        - 소스 코드에서 명령된 연산들이 CPU에서 실행
    - 메모리 할당
        - 동적으로 메모리를 할당하는 작업 등
    - 예외 처리
        - 오류 메세지 등으로 알리거나 대처
    - 입출력 작업
        - 파일 읽기/쓰기, 네트워크 통신, 사용자 입력 등 외부와의 상호작용
    - ex) 0으로 나누기, 잘못된 메모리 접근, 배열 인덱스 초과 등

50. **Heap과 Stack의 메모리 할당에 대해 설명하시오**
- 높은 주소 -> 낮은 주소 순서
- 스택 영역
    - 클래스의 멤버변수, 함수의 매개변수, 지역변수, 함수의 반환주소값
    - 함수 호출 시 자동으로 할당되는 메모리 (동적)

        - 컴파일 타임에 크기가 결정
        - 함수가 끝나면 자동으로 해제 (하나씩 pop 해가면서 반환주소를 찾아간다)
        - 스레드가 공유하지 않음, 독립적
    - 힙 영역
        - 동적으로 할당되는 메모리
        - 프로그래머가 직접 할당과 해제 관리해야 함
        - 주로 큰 데이터를 저장할 때 사용
        - 런타임에 크기 결정
        - 스레드가 공유
    - 데이터 영역
        - 전역 변수, 정적 변수
        - 프로그램이 시작될 때 메모리에 할당, 종료될 때 까지 유지
        - 프로그램 실행될 때 크기 결정
        - 모든 스레드가 공유
            - 전역 변수
                - 프로그램 전체에서 접근 가능
                - 파일 최상위 수준에서 선언
                - 수명 : 프로그램의 시작과 함께 할당, 종료와 함께 해제
                - 할당 순서가 미정, 타이밍에 주의해야함
                - 해제 순서는 할당의 역순
                - 범위 : 프로그램 전체
            - 정적 변수
                - 수명은 프로그램 전체 수명으로 유지하되, 범위는 제한
                - 프로그램 시작, 종료와 함께 생성, 해제
                - 첫번째 호출 시 한번만 초기화
                - 함수 내 정적 변수
                    - 함수 호출될 때 초기화
                    - 함수 내에서만 접근 가능
                    - 여러번 호출되어도 이전 값 유지
                    - 함수가 끝나도 값이 유지
                - 파일 내 정적 변수 (파일 범위)
                    - 파일 전역 범위에서 선언
                    - 파일 내부에서만 접근 가능
                    - 외부와 분리되어 캡슐화된 형태로 사용됨
    - 코드 영역
        - 실행될 코드가 저장됨
        - 읽기 전용 메모리
        - 읽기 전용이지만 스레드, 프로그램이 공유 가능
        - 읽기 충돌 발생하지 않음
        - 복사본 여러개 만들 필요 없음
        - 프로그램이 시작할 때 고정된 메모리 위치에 로드됨
            - 실행이 종료될 때까지 변하지 않음
            - 프로그램의 수명동안 고정
            - (실행 중 동적으로 할당/해제되는 다른 메모리 영역 : 힙, 스택)

53. **깊은 복사와 얕은 복사에 대해 설명하시오**
    - 얕은 복사

        - 객체의 참조(메모리 주소)만 복사하여 원본과 복사본이 같은 데이터(같은 메모리 주소에 있은 값, 동일한 자원)를 참조
        - 하나의 객체가 소멸될 때 동적 메모리가 해제되면, 다른 객체가 포인터는 댕글러 포인터가 되어 잘못된 동작 일으킬 수 있음
    - 깊은 복사 Deep Copy
        - 객체의 실제 데이터를 복사
        - 주소가 가리키는 동적 메모리 공간까지 복사
        - 원본과 복사본이 서로 다른 데이터(서로 다른 메모리 주소)를 가리키게 됨
        - 더 많은 메모리를 사용하지만 독립적인 데이터 보장
            - 하나가 소멸 될 때 다른 데이터에 영향 없음
            - 메모리 누수 등 방지

54. **L-Value와 R-Value에 대해 설명하시오**
- L-Value
    - 할당이 가능한 메모리 주소를 가지는 값, 메모리 위치
    - 대입 연산자의 왼쪽에 올 수 있음
- R-Value
    - 임시 값을 의미, 메모리 위치 없음, 수명 짧음
    - 대입 연산자의 오른쪽에 올 수 있음
    - 연산 결과 등
- std::move
    - vector<int> v1
    - vector<int> v2 = std::move(v2);
        - R-value로 변환해, 데이터를 복사하는 대신 이동 시킴
        - 메모리 복사 비용 절약 가능
        - shift 개념

55. **소멸자에 virtual을 붙여야 하는 이유에 대해 설명하시오**
- 상속 관계에서 기본 클래스의 소멸자로 virtual로 선언하지 않으면, 파생 클래스의 소멸자가 호출되지 않아 메모리 누수 발생 가능
- virtual 사용하면 파생 클래스의 소멸자도 함께 호출되어 메모리를 안전하게 해제할 수 있음

56. **다중 상속에 대해 설명하시오**
- 다중 상속
    - 하나의 클래스가 두개 이상의 클래스를 상속 받는 것
    - 이를 통해 여러 클래스의 기능을 조합할 수 있음
    - 하지만 같은 이름의 함수나 멤버 변수가 여러 클래스에 존재할 경우 모호성이 발생 가능
        - 다이아몬드 문제
            - 두 부모 클래스가 같은 조상을 가질 때 중복된 멤버 상속 가능
        - 이를 해결 하기 위해 가상 상속을 사용 가능
- class C : public A, public B
- 가상 상속
    - class A : public virtual Base
    - class B : public virtual Base

57. **추상클래스를 포인터형으로 사용 가능한가**

- 추상클래스는 포인터형으로 사용 가능
- 추상 클래스 자체는 인스턴스화(객체화)가 불가능
- 추상 클래스를 상속받은 클래스의 객체를 가리키는 포인터로 사용할 수 있음
    - 추상클래스* ptr = &자식클래스 (애초에 이 오른쪽 부분이 자기 자신일 수는 없음. 오른쪽은 객체여야 하기 때문에)
- 다형성
    - 상속받은 클래스의 객체를 기본 클래스의 포인터나 참조로 다룰 수 있는 성질
    - 다형성의 핵심은 “동적 바인딩”
        - 컴파일 시점이 아닌 실행 시점에 호출될 함수가 결정됨
        - 같은 기본 클래스 포인터로 다양한 파생 클래스 객체 처리 가능

58. **애플리케이션에서 함수가 메모리 상에서 동작하는 원리에 대해 설명하시오**
- 함수는 코드 메모리 영역에 저장
- 함수가 호출되면 스택에 해당 함수의 프레임 생성
    - 프레임에는 함수의 매개변수, 지역변수, 리턴주소 등이 저장
- 함수가 종료되면 스택 프레임이 해제되고 리턴 주소로 복귀

59. **트리의 개념을 설명하시오**
- 트리
    - 계층 구조를 가지는 자료구조
    - 노드와 노드 간에 부모 - 자식 관계가 있음
- 루트 노드 = 최상위 노드
- 리프 노드 = 자식 노드가 없는 말단 노드
- 탐색, 삽입, 삭제 등 사용
- 이진 트리
- 이진 탐색 트리
- AVL 트리
    - 자가 균형 이진탐색트리의 일종
    - 균형을 엄격히 유지
        - 항상 O(logn) 보장
        - 탐색, 삭제, 삽입 모두 동일
    - 다른 자가 균형과 비교
        - 레드블랙 트리
            - AVL보다 균형이 덜 엄격
            - 삽입, 삭제 시 회전 연산이 더 적어, 삽입과 삭제 빈도가 높은 경우 레드블랙 효율적
        - B-트리
            - AVL보다 노드 당 여러개 값 저장 가능
            - 대용량 데이터에서 더 적은 디스크 접근으로 관리 가능

60. **이진트리의 개념을 설명하시오**
- 이진 트리
    - 각 노드가 최대 두개의 자식을 가지는 트리
- 이진 탐색 트리
    - 왼쪽 자식 노드 < 부모
    - 오른쪽 자식 노드 > 부모

- 삽입, 삭제, 탐색 작업에서 평균적으로 O(log n) 의 시간복잡도 가짐

61. **레드블랙 트리의 개념을 설명하시오**
- 레드블랙 트리
    - 이진 탐색 트리의 일종
    - 자가 균형 이진탐색트리
    - 노드가 빨간색 또는 검은색
    - 트리의 높이를 균형있게 유지하기 위해 몇가지 규칙 있음
        - 무슨 규칙?
        - 이를 통해 최악의 경우에도 O(logn)의 성능 보장

63. **퀵 소트의 시간복잡도에 대해 설명하시오**
- 퀵소트
    - 평균 시간 복잡도 O(nlogn)
    - 피벗을 기준으로 배열을 두 부분으로 나누고, 각각 재귀적으로 정렬하는 방식
    - 최악의 경우, 즉 이미 정렬된 경우
        - O(n^2)

64. **STL이란?**
- Standard Template Library
- C++에서 제공하는 라이브러리
- 다양한 자료구조와 알고리즘 포함
- 벡터, 리스트, 맵 등과 같은 컨테이너
- 이를 다룰 수 있는 알고리즘, 반복자 등 제공

65. **벡터의 재할당(Resize)란 무엇인가?**
- 백터의 재할당 (Resize)
    - 벡터는 동적 배열
        - 현재 크기보다 많은 데이터를 저장하려 할 때 메모리를 다시 할당하여 크기를 늘림
        - 이 과정에서 기존 데이터를 새로운 메모리 공간으로 복사하고, 배열의 크기를 확정
- 재할당은 비용이 크므로, 미리 적절한 크기로 할당하거나, reserve 함수를 사용해 재할당을 줄일 수 있음

66. **맵과 해쉬맵의 차이점을 설명하시오**
- 맵
    - 이진 탐색 트리 기반의 컨테이너
    - 키가 정렬된 상태로 저장됨
    - 평균 O(logn)의 탐색 시간
    - 자가 균형 이진탐색트리 (레드-블랙 트리)로 구현되어 있음
    - 키-값
        - 키로 검색하여 값을 찾음
    - 이진 탐색을 사용하므로 검색이 빠른 편
        - 삽입, 삭제 시 균형트리이기 때문에 균형을 유지하는데 오버헤드가 생긴다
    - 맵에 삽입 하는 경우

        - 이미 있는 키라면 값을 삽입하지 않고 이미 키가 있는 해당 노드의 키/값을 리턴
        - 정확히는 맵은 insert를 할 경우 무조건 pair를 리턴하는데
            - pair<pair(key, value), bool>를 리턴
            - bool은 해당 키가 이미 맵에 있는지 없는지.
- 해쉬맵, 언오더드맵
    - 해쉬 테이블 기반
    - 키를 해시함수를 통해 특정 값(보통 integer)으로 변환하여 저장
    - (탐색 빠름) 해시 함수로 인한 평균 O(1)의 상수 탐색 시간
    - 단, 해쉬맵은 키의 순서를 보장하지 않음(정렬X)
    - C++ 11 이전엔 정렬이 필요하지 않은 경우에도 map을 사용하여 불필요한 오버헤드를 감수해야 했다. 또는 비표준 라이브러리의 hash_map 컨테이너를 사용해야 했다.
    - 삽입, 삭제도 평균 O(1)
    - 여유 공간이 실제 공간보다 많이 필요
    - 공간을 너무 적게 잡으면 충돌이 자주 일어남
    - 해시맵을 구현하는 방법으로는 개방 주소법, 체이닝 등
    - 체이닝 (Chaining)
        - 동일한 해시 값을 가진 요소들을 하나의 리스트(혹은 연결리스트)로 저장
        - 장점
            - 충돌이 많아져도 해시맵의 각 슬롯이 리스트로 확장되므로 메모리 공간 유동적 사용 가능 (동적 관리)
        - 단점
            - 충돌이 많아질 경우, 하나의 슬롯에 리스트가 길어져 탐색 시간이 느려질 수 있음 (최악의 경우 O(n))
    - 개방주소법 (Open Addressing)
        - 충돌이 발생했을 때, 다른 빈 슬롯을 찾아 데이터를 저장
        - 체이닝과 달리 모든 데이터를 해시맵의 슬롯에 저장
        - 탐색 시에도 동일한 방식으로 해시함수로 계산한 인덱스부터 시작해 찾아나감
        - 장점
            - 추가적인 리스트를 사용하지 않아 메모리 효율 좋음
            - 해시맵 전체를 하나의 연속된 배열로 관리 가능
        - 단점
            - 충돌이 많이 발생할 경우 빈 슬롯을 찾기 위한 탐색 비용 커질 수 있음 (최악의 경우 O(n))
            - 해시맵이 거의 다 찰 경우 충돌 해결이 어려워지고 성능 저하 가능성