

보물고블린 초판

본 문서는 면접과 필기시험을 여러 번 보면서 뚝배기가 깨져 본 결과 신입 면접은 어느정도 패턴을 공유한다는 것을 알게 되었으므로, 패턴을 공유하기 위해 만들었습니다.

이 문서는 본인이 면접/필기시험 시 답을 하기 위한 공부를 한 기록을 적당히 읽기 좋게 고친 문서이므로 개념설명보다는 실제 나온 문제를 제시하는 경우가 많고 답도 대충 써있거나 없는 경우도 있습니다. 의문이 가면 직접 코드를 짜 보거나, 찾아보는 작업을 하는 것이 좋습니다.

기본적으로 내가 공부하려고 끄적인 것들을 모아놓은 것이므로, 다른 사람이 보기에 이해가 안되는 서술이 있을 수 있으므로 그런 건 본인에게 물어보시면 됩니다.

목차

1. 면접, 필기시험 준비를 위한 공부내용 정리	4
1.1 객체지향과 C++	4
1.1.1 절차적 프로그래밍과 객체지향의 차이점	4
1.1.2 객체지향의 장점	4
1.1.3 객체지향의 특징	4
1.1.4 클래스	6
1.1.5 다형성	9
1.1.6 포인터와 참조자, Malloc과 New	10
1.1.7 가상함수	17
1.1.8 C++ 스타일 캐스팅	19
1.1.9 STL 컨테이너	21
1.1.10 가비지 컬렉터	24
1.1.11 타입 간의 계산 결과	25
1.1.12 생성자와 소멸자	26
1.1.13 C++의 메모리 영역	30
1.1.14 C++11/14에 추가된 신기능	30
1.1.15 const	31
1.2 컴퓨터공학	32
1.2.1 프로세스와 스레드	32
1.2.2 데드락	33
1.2.3 메모리	35
1.2.4 네트워크	41
1.3 컴퓨터 그래픽스	43
1.3.1 렌더링 파이프라인	43
1.3.2 조명	45

2. 실제 면접 질문 사례 모음	47
2.1 1차면접 모음	47
2.1.1 N모사 수시채용(RPG)	47
2.1.2 M모사 수시채용(PC, IP기반 신작 RPG)	48
2.1.3 D모사 수시채용(퍼즐, 모바일)	48
2.1.4 A모사 수시채용(액션 RPG)	48
2.1.6 N모사 수시채용(모바일 IP RPG)	49
2.1.7 N모사 공개채용(액션 RPG)	50
2.1.8 V모사 수시채용(모바일)	50
2.1.9 A모사 수시채용(액션 RPG)	50
2.1.10 P모사 수시채용(PC RPG)	51
2.1.11 K모사 공개채용(PC RPG)	52
2.2 2차면접 모음	52
2.2.1 A모사 수시채용(액션 RPG)	52
2.2.2 N모사 공개채용(액션 RPG)	53
2.2.3 M모사 수시채용(PC, IP기반 신작 RPG)	53
3. 주관적 팁 모음(주관성, 비정확성 주의)	54
3.1 1차면접의 패턴	54
3.1.1 경험을 묻는 질문의 예시	54
3.1.2 이러한 질문에 대처하려면	55
3.1.3 지식과 테크닉을 묻는 질문의 예시	55
3.1.4 이러한 질문에 대처하려면	56
3.3 그 외 팁	56

1. 면접, 필기시험 준비를 위한 공부내용 정리

1.1 객체지향과 C++

한줄평 : 면접관이 시험으로도 질문으로도 제시하기 만만한 분야입니다. 그만큼 자주 나옵니다. 정의를 암기해 가는 것이 좋습니다.

메서드 == 함수 입니다.

1.1.1 절차적 프로그래밍과 객체지향의 차이점

절차적 프로그래밍(Procedural Programming)

프로그램 설계 시 기능 구현을 위해 프로시저(함수)를 중심으로 사용하여 구조/로직을 설계하는 방법

객체지향 (Object-Oriented Programming (OOP))

프로그램 설계 시 프로그램을 수많은 객체로 나누고 이 객체들의 상호작용으로 서술하는 방법

1.1.2 객체지향의 장점

코드 재사용이 용이함. 클래스를 기능별로 분할하였기에 모듈화도 용이하고 상속을 통해 높은 확장성을 가질 수 있다. 객체를 만들어두면 재사용할 수 있다. 그렇기 때문에 유지보수성이 뛰어나다. (게임은 늘 갈아엎고 뒤집히는 경우가 많기 때문에 유지보수성에 특히 민감하다)

1.1.3 객체지향의 특징

캡슐화(encapsulation)

클래스를 통해 변수와 함수를 하나의 단위로 묶는다.

하나의 기능을 하는 요소들을 모두 한 캡슐에 모아둔 것 같다고 하여 캡슐화이다.

같은 역할을 하는 변수, 함수들을 모아두었기 때문에 의존성, 커플링이 줄어든다. 그로 인해 관리가 용이해진다.

정보은닉(Information hiding)

프로그램의 세부 구현을 감추는 것.

private 접근자를 통해 변수와 객체를 감추고, **public** 접근자로 선언된 메소드(함수) 로만 접근을 허용하여 접근을 제한하는 것.

왜? 의도하지 않은 접근을 제한하기 위하여 (= 설계한 클래스를 다른 사람이, 혹은 본인이 잘못 쓰는 것을 방지하기 위하여.)

ex) 다른 사람이 설계한 클래스의 변수에 직접 접근하여 값을 바꾸는 것을 막는다.

이러면 코드 사용자는 프로그램의 내부 구현에 신경 쓸 필요 없이 메소드를 통해 원하는 기능을 호출하기만 하면 된다.

□ 제대로 쓰기엔 쉽게 설계하고, 잘못 쓰기엔 어렵게 설계하라.

또한, 객체들 끼리 서로의 구현과 상태를 상관하지 않고 단지 사용하게만 함으로써 의존성을 낮추는 역할을 한다.

상속(Inheritance)

자식 클래스가 부모 클래스의 변수, 함수를 물려받는 것.

자식 클래스는 자신만의 특성을 오버라이딩을 통해 디테일하게 구현한다.

왜? 다형성과도 연관되는 질문인데, 부모와 자식을 다형성으로 묶어놓으면 부모에서 추상적인 형태를 주고, 자식에서 세밀화하여 유연하게 코드를 작성할 수 있으며 생산성, 유지보수성 또한 올라간다. (유지보수성의 경우 무분별한 복사-붙여넣기를 막을 수 있는 효과가 있다.)

(복붙으로도 간단하게 상속으로 할 수 있는 일을 할 수 있잖아? 라는 후질문이 들어오면) -> 또한 나중에 기능 개선 시 '교체'가 가능한 장점이 있다. ex) A사의 타이어 객체를 사용하고 있다가, B사의 타이어 객체로 개선하고 싶으면 B사는 '타이어'를 부모로 상속받아 자신에 맞게 구현하기만 하면 된다.

카피-페이스트로 만들어진 코드들은 각각이 다른 코드이므로 개선이 필요할 때 구현을 따로따로 해줘야

하며, 유지보수가 어려워진다.

추상화(abstraction)

공통의 속성이나 기능을 묶어 이름을 붙이는 것

1.1.4 클래스

클래스란?

C++에서 객체를 구현하기 위해 사용하는 방법이다.

객체의 상태/특성을 정의하는 일종의 설계도 (청사진, **Blueprint**)이며, 객체화했을때 (인스턴스화 라고 한다) 객체가 된다.

즉 클래스 != 인스턴스이며, 클래스가 인스턴스화 해야 인스턴스(객체)가 될 수 있다.

변수와 메서드를 가진다.

인스턴스

클래스를 기반으로 만들어져 실제로 메모리에 할당된 것

구조체/ 클래스 패딩

패딩이란 구조체나 클래스를 실제로 메모리에 올릴 때, 성능 향상을 위해 추가적인 메모리를 할당하여 끼워 넣는 것을 의미한다.

패딩을 하는 이유(인터넷 펌)

메모리에서 CPU 레지스터로 한번에 읽어오는(fetch) 데이터의 크기 때문에 패딩이 일어납니다. 32비트 머신에서는 4바이트 씩이고 64비트 머신에서는 8바이트 씩이겠지요.

이렇게 선언을 하면 메모리 맵상의 어딘가에 영역이 잡히겠죠

그런데 패딩이 없다고 가정하면,

|a|bb|cccc|a|bb|cccc|a|bb|cccc|a|bb|cccc|...

이런식으로 잡히겠죠...

fetch는 4바이트 단위로 이루어지니깐 만약 첫번째 **c**를 접근하기 위해서는

2번의 **fetch**가 이루어져야 합니다.

0번째 바이트부터 3번째 바이트까지 4바이트를 읽어서 그중에 3번째 바이트를 먼저 취하고,

4번째 바이트부터 7번째 바이트까지 4바이트 읽어서 그중에 4~6번째 바이트를 취해서,

둘을 합해 최종 **c**의 값을 만드는 결과가 생기지요~

때문에 컴파일러가 패딩을 넣는 겁니다.

한줄요약 : 구조체/클래스 패딩은 구조체 안의 가장 큰 자료형의 크기로 정해진다.

만약 구조체 안에 구조체가 들어가 있다면, 그 구조체 안에서 가장 큰 자료형의 크기로 정해진다.

만약 설정이 **8byte** 패딩으로 되어 있는데 구조체가

```
class C
{
    short a;
    int b;
    int c;
};
```

이와 같이 구성되어 있다면, **short-int**를 하나로 묶어서 **6바이트 + 2바이트 패딩(총 8바이트)**

int + 패딩 4바이트 (총 8바이트)

식으로 묶는다. (단, short는 2bytes, int는 4bytes일 때의 풀이이다)

만약 설정이 4bytes 패딩이라면,

```
short a; //4bytes
```

```
int b; //4bytes
```

```
int c; //4bytes
```

총 12바이트가 된다.

이름	값
sizeof(C)	12

예시

```
struct Name
{ //2
    char name[2];
};
```

--□ 답은 2이다. 왜냐면 패딩은 레지스터 단위 접근을 쉽게 하기 위해서 만드는 것인데 char[2] 는 2바이트이며, 다음에 접근할만한 요소가 없기 때문에 2만 차지해도 되기 때문이다(만약 int형 변수가 뒤에 하나 더 선언되어 있다면 총 용량은 2 + 2(패딩) + 4였을 것이다.)

```
struct Point
{ //8
    short point_x;
    int point_y;
};
```

//short + 2byte / int

```
struct Player
{ //20
    char rank;
    short hp;
    short mp;
```



```

    Point point;
    Name name;
};

```

Point의 **제일 큰 타입이 int**이므로, 4byte 기준으로 패딩이 잡힘.
char + short + 1byte 패딩 / short + 2byte 패딩 / 8byte(point) / 4byte (name)
mp와 point는 따로 용량을 잡는다.

```

struct long_Point
{ //16
    double point_x;
    int point_y;
};

```

double = 8bytes이므로 총 메모리 용량은 **8+8 = 16bytes**

소켓 프로그래밍 중 패킷 구조체를 구현할 때 **#pragma pack(push, 1)**을 쓰는 이유도

패딩에 있다.

패킷을 받은 쪽은 패킷을 읽어서 값을 복구해야 하는데(캐스팅을 통해), 패딩이 들어가 있으면 잘못된 크기를 가지고 캐스팅을 하게 되므로 잘못된 값이 복구되기 때문이다.

1.1.5 다형성

다형성

객체지향에서 다형성은 여러 가지 형태를 가질 수 있는 능력을 의미함.

C++에서의 다형성은 상속에 의한 다형성, 오버로딩, 오버라이딩, 템플릿에 의한 다형성 등으로 구현된다.

오버로딩 -> 같은 이름의 함수라도 인자(Parameter) 타입이 다르거나, 개수가 다르면 다른 함수로 정의하여 사용할 수 있음. (주의 : 리턴 타입은 오버로딩과 아무 상관이 없음. 리턴 타입으로 함수를 구분할 수 없기 때문)

오버라이딩 -> 상속 관계에서 부모의 메서드를 자식이 자기에 맞게 재정의 해서 사용함

템플릿 -> 클래스, 함수를 타입에 독립적이게 만드는 도구입니다. 템플릿을 사용하면

여러 타입에 대응되는 단 하나의 객체나 함수를 만들 수 있습니다.

템플릿의 타입은 컴파일 타임에 결정되어 인스턴스화 하기 때문에 컴파일타임에 컴파일러가 실제 코드 구현부를 모두 볼 수 있어야 합니다.

인스턴스화는 컴파일러가 컴파일 시에 요구되는 타입으로 클래스 정의 코드를 생성해내는 것입니다. 즉 타입을 지정하여 그 타입을 사용하는 클래스를 만들어냅니다.

(ex. 만약 float타입으로 템플릿 클래스를 사용했다면 컴파일 타임에 컴파일러가 float 타입의 클래스를 만들어낸다, 이를 인스턴스화라고 함)

그렇기 때문에 템플릿 클래스의 함수 구현은 헤더 파일에 들어가야 합니다.

혹은 헤더 파일에서, 템플릿 정의부분 아래에 cpp파일을 인클루드 하는 방법도 있습니다. 이 방법을 사용할 때 주의할 점은 cpp파일을 빌드 리스트에 넣지 말아야 한다는 것입니다.

1.1.6 포인터와 참조자, Malloc과 New

참조자(reference)

실체가 있어야 하며 선언 즉시 할당되어야함. 즉 NULL, nullptr 로 할당 불가능.

레퍼런스는 초기화리스트를 사용하여 먼저 초기화해야하는데(modren c++의 초기화리스트와는 다름. 생성자의 초기화리스트를 의미함), 이는 생성자 내부에서의 초기화는 먼저 null로 생성한 뒤 값을 넣는 방식이기 때문. 또한 한번 할당하면 다른 곳에 재할당 불가능

포인터(Pointer)

실체가 없이 NULL이 가능하며, 언제든지 할당할 수 있음. 또한 동적 메모리 할당에 사용함.

참고로 초기화리스트는 상수, 참조자, has-a관계의(포함한) 클래스 초기화 에 사용해야 한다.

Malloc과 New의 차이점(C 스타일 동적 할당과 C++ 스타일 동적 할당의 차이점)

Malloc : 단순한 메모리 할당. 할당 시 메모리의 사이즈를 입력해서 할당받음. C스타일

malloc은 void* 를 리턴하기 때문에 원하는 타입으로 캐스팅해서 사용

New : 할당과 동시에 초기화 가능(초기값을 줄 수 있음).

생성자 호출됨(즉 C++ 객체 할당에 사용함). 오버로딩 가능(new도 연산자이다)

할당 시 객체의 크기를 입력하여 할당받음. C++ 스타일

```
int * c_style = (int*)malloc(sizeof(int) * 10);  
int * cpp_style = new int[10];
```

댕글링 포인터

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.

그래서 메모리를 해제하는 구문 이후, 해당 포인터 변수를 nullptr로 바꿔주고 사용할 때마다 nullptr인지 체크하는テクニック이 필요하다.

비주얼 스튜디오의 경우, 메모리를 해제하면 0x0823같은 메모리로 치환하여 강제 크래시를 유발하기도 한다

(의도적인 크래시가 미정의 동작보다 차라리 안전하기 때문이다)

```

int *g = nullptr;

int main()
{
    int *p = new int;
    *p = 7;
    g = p;

    //...
    delete p;
    p = nullptr;
    //...

    *g = 4; //올바르지 않은 메모리!
}

```

단, 이런 문제가 발생할 수 있다. 아무리 `p`를 할당 해제 후 `nullptr`로 치환하였다고 해도, `p`를 참조하는 포인터 변수가 있었다면(`g`와 같은) 이는 추적하기 힘든 메모리 누수/버그를 발생시킬 수 있다. 스마트 포인터를 쓰면 이러한 문제를 어느정도 해소할 수 있다.

스마트 포인터

이전의 C++에서 동적 메모리를 할당하기 위해서는 `new/delete`를 이용해 포인터 변수를 정의하여 사용하여야 했다.

이러한 포인터를 원시 포인터(날 포인터, raw pointer)라고 한다.

C++에서의 메모리는 프로그래머가 책임지고 `delete`를 통해 사용되지 않는 메모리를 반환해야 하였고, 메모리를 반환하지 않아 메모리 누수(memory leak)가 발생하거나 적절하지 못한 타이밍에 메모리를 `delete`하여 댕글링 포인터(dangling pointer)를 발생시켜 프로그램의 안전을 심히 위협하는 경우가 생기기도 하였다. 특히 메모리 버그는 찾기 겁나게 어려우므로 항상 문제가 되었다.

스마트 포인터는 C++11에서 공식적으로 추가된 기능으로, 스마트 포인터를 사용하면 객체가 더는 필요하지 않을 때 객체에 할당된 메모리가 자동으로 해제된다. 즉 메모리 누수의 가능성을 영구적으로 제거한다.

C#이나 Java처럼 Garbage Collector를 사용하는 것은 아니고 레퍼런스 카운트를 통하여 제거 시점을 결정한다.

스마트 포인터는 <memory> 헤더 안에 정의되어 있다.

스마트 포인터와 원시 포인터의 차이점

1. 스마트 포인터는 자유 공간(new/delete로 할당한 공간, malloc/free로 할당한 공간은 힙 공간이라고 한다. 서로 크게 다른 개념은 아닌 것 같다.)에 할당한 메모리의 주소만 저장할 수 있다.
2. 원시 포인터에서 하던 증가, 감소 같은 산술 연산은 스마트 포인터에서 할 수 없다.

unique_ptr<T>

unique_ptr<T>는 타입 T에 대한 포인터처럼 사용되며, 언제나 유일해야 한다. 즉 둘 이상의 unique_ptr이 같은 주소를 가질 수 없으며, unique_ptr은 복사될 수 없다. (유일성을 손상시켜서는 안 된다)

unique_ptr을 옮기려면 오직 소유권을 이동시키는 행위만이 허용된다.

utility헤더에 정의된 std::move()를 사용하면 unique_ptr객체에 저장된 주소를 다른 unique_ptr객체로 이동시킬 수 있으며, 소유권을 이전하면 기존의 unique_ptr 객체는 무효화(Invalidate)된다.

그러므로, 객체에 대한 단일 소유권만 허용하고 싶을 때는(반드시 하나만 가지고 사용하고 싶을 때는) unique_ptr<T>를 이용하는 것이 좋다.

unique_ptr의 사용방법

```
std::unique_ptr<type> name {new type()};
```

ex)

```
std::unique_ptr<std::string> pname {new std::string {"Algernon"}};
```

```
auto pname = std::make_unique<std::string>("Algernon");
```

```
auto pstr = std::make_unique<std::string>(6, '*');
```

스마트 포인터는 원시 포인터와 마찬가지로 역참조를 통해 객체에 접근할 수 있다.

```
std::cout < *pname << std::endl;
```

임의의 크기로 스마트 포인터 배열을 생성할 수도 있다.

```
int len = 10;
```

```
std::unique_ptr<int[]> pnumbers{new int[len]};
```

```
std::unique_ptr<int[]> pnumbers = std::make_unique<int[]>(len);
```

`unique_ptr` 객체는 복제가 불가능하므로, 함수에 값으로 전달할 수 없다. 만약 `unique_ptr` 객체를 함수의 인수로 쓰고 싶다면 인수를 참조 매개변수로 받아야 한다.

`unique_ptr` 객체는 복제될 수 없지만 암묵적 이동 연산(implicit move operation)에 의해 반환이 가능하므로 함수에서 반환될 수 있다.

reset() -> 스마트 포인터가 가리키는 원본 객체를 소멸, `unique_ptr`은 해제된다.

`ptr.reset(new std::string{"Fred"})`와 같이, **reset**의 인자에 새 객체를 넣으면 스마트포인터와 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 비교한다.

shared_ptr<T>

shared_ptr<T>객체는 타입 T에 대한 포인터처럼 행동한다.

unique_ptr과는 반대로 shared_ptr<T>는 객체를 여러 shared_ptr<T>와 공유할 수 있다.

shared_ptr<T>는 레퍼런스 카운팅 방식을 사용하여 메모리를 관리하는데 새로운 shared_ptr<T>객체가 주소를 공유받을 때 마다 레퍼런스 카운트가 증가하며, 공유를 받았던 shared_ptr 객체가 소멸되거나, 다른 주소를 할당받거나, nullptr를 할당받으면 해당 객체의 레퍼런스 카운트가 감소한다. 만약 레퍼런스 카운터가 0이 되면(해당 주소를 가리키는 shared_ptr<T>객체가 없으면) 해당 주소를 위한 힙 메모리가 자동으로 해제된다.

새로운 shared_ptr을 정의하면 두 가지 할당을 받게 된다.

첫 번째는 shared_ptr이 가리키는 원본 객체를 위한 힙 메모리를 할당하게 된다.

두 번째는 스마트 포인터의 레퍼런스 카운트를 위한 컨트롤 블록을 위해 스마트 포인터 객체와 관련된 힙 메모리를 할당받게 된다.

shared_ptr의 사용방법

std::shared_ptr<double> pdata {new double(999.0)}; 일반적인 할당 방법

std::shared_ptr<double> pdata = std::make_shared<double>(999.0); 일반적인 할당보다 효율적으로 동작한다. 추천되는 방법이다.

std::shared_ptr<double> pdata2{pdata}; 다른 shared_ptr로 초기화하는 스마트 포인터. 레퍼런스 카운트가 증가한다.

double *pvalue = pdata.get() 스마트 포인터의 원시 포인터 객체를 반환한다. 반드시 사용해야 할 경우가 있을 때만 이렇게 사용해야 하며, get()을 통해 반환한 원시 포인터를 이용해서 shared_ptr<T>를 생성하는 것은 올바르지 못한 행동이며 위험을 초래할 수 있다.

```
pname.reset(new std::String{"Jane Austen"});
```

reset()을 인수없이 호출하면 레퍼런스 카운트가 1 감소하며, 해당 **shared_ptr** 객체가 아무것도 가리키지 않게 된다.(**unique_ptr**의 동작과는 좀 다르다, 물론 레퍼런스 카운트가 1이었다면, 해당 **shared_ptr** 객체를 **reset**하는순간 스마트 포인터가 해제된다.)

reset()의 인수로 원시 포인터를 전달하면 **shared_ptr**이 가리키는 주소를 바꿀 수 있다.

shared_ptr끼리의 비교는 **==** 연산자를 사용하여 비교할 수 있다. 단 두 객체가 모두 **nullptr**일 수 있으므로, 같은 객체를 가리키는지만 체크하면 안 된다.

스마트 포인터는 암묵적으로 **bool**로 변형될 수 있으므로(객체가 있으면 **true**, **nullptr**이면 **false**) **if((pA == pB && (pA != nullptr))** 혹은 **if((pA == pB && pA))**로 비교할 수 있다.

pname.use_count() -> **use_count()**는 해당 **shared_ptr**이 가리키는 객체의 레퍼런스 카운트를 반환한다.

shared_ptr이 **nullptr**을 가리키고 있다면 0을 반환한다.

.unique() 함수는 **shared_ptr**이 유일한지(**true**), 복제본이 있는지(**false**) 확인할 수 있다.

weak_ptr<T>

weak_ptr는 **shared_ptr**의 상호참조 문제를 해결할 수 있다.

weak_ptr은 **shared_ptr**객체에서 생성하여 연결하며, 같은 주소를 가리킨다.

단 **weak_ptr**은 연결된 **shared_ptr**객체의 레퍼런스 카운트를 증가시키지 않으므로, 객체의 소멸에 관여하지 않는다.

shared_ptr의 메모리가 레퍼런스 카운트가 0이되어 해제되더라도 연관된 **weak_ptr** 객체는 남아있게 된다.

1.1.7 가상함수

추상 클래스

추상 클래스는 보다 구체적인 클래스가 파생될 수 있는 일반 개념식 역할을 합니다.

C++에선 순수 가상함수가 하나라도 포함된 클래스는 추상 클래스로 변환됩니다.

추상 클래스는 인스턴스화 할 수 없습니다.

순수 가상함수는 내용이 없이 형식만 정의된 함수입니다. ‘이 형식대로 만들어서 사용하라’는 의미로, 자식 클래스는 이를 반드시 오버라이드 해야 합니다.

```
virtual void Func() = 0;
```

순수가상함수는 이렇게 선언합니다.

virtual 키워드

가상 함수임을 나타내는 키워드이며, 상속 관계에서 자식이 해당 함수를 오버라이딩 하기 위해 사용한다. 즉 부모 함수에 **virtual**이 없으면 자식이 오버라이드 할 수 없다. 또한 **virtual** 키워드를 사용했을 시 **런타임에 가상함수 테이블이 생성되어** 올바른 함수를 찾아갈 수 있게 만든다. 상속 관계에서 부모의 소멸자엔 반드시 **virtual**을 붙여야 하는데, 이것은 소멸 시에 **virtual**이 없으면 올바른 자식 객체까지 찾아가지 못 하여 부모의 부분만 소멸되고 자식의 부분은 남는 현상이 발생하기 때문이다.

(이는 곧 메모리 누수, 오염으로 이어진다)

객체 생성 및 소멸 과정에서는 절대 생성자/소멸자에서 가상 함수를 호출하면 안 되는데,

자식 객체 생성 시에 가상함수를 호출하게 되면 부모가 먼저 생성될 때, 자식 객체는 아직 초기화되지 않은 상태이므로 자식 객체는 자신이 부모 클래스인 것처럼 동작한다. 즉 부모 클래스의 **virtual**이 호출된다. 그래서 의도하지 않은 동작을 하게 된다.

또한 미정의 동작을 수행할 수 있다. (**Undefined behavior**)

가상 함수 테이블

가상 함수 테이블은 클래스마다 존재한다.(인스턴스마다가 아님)

각각의 인스턴스들은 해당 클래스의 가상함수 테이블을 가리키는 포인터 변수를 하나씩 가진다.

즉 가상함수가 있는 클래스의 인스턴스 메모리 크기를 구할 땐 이 포인터 변수의 크기를 더해주는 것을 잊지 말아야 한다.

(32bit 환경이면 4bytes, 64bit 환경이면 8bytes)

상속 관계의 클래스들은 가상함수 테이블을 사용해 오버라이드 된 함수를 찾아간다.

가상함수테이블은 해당 클래스의 모든 가상함수의 주소값을 제공한다.

vtable에는 해당 가상함수들의 주소값이 들어있다. 상속 관계가 **A->B**면

클래스 **A**의 **vtable**에는 **A**의 가상함수들

A::aaa(void)

클래스 **B**가 만약 **A**의 **aaa**함수를 오버라이딩 했다면 **B**의 **vtable**에는

B::aaa(void) 식으로 가상함수 테이블의 내용이 작성된다.

실제 코드로 보면

```

class Parent
{
public:
    int a;

    virtual void Func() {}
    virtual void FF() {}
};

class Child : public Parent
{
public:
    void Func() override {}
};

```

p	{a=-858993460 }	Parent
__vfptr	0x00007ff6898644f0 { 연습용.exe!void(* Parent::`vftabl...	void **
[0]	0x00007ff689851726 { 연습용.exe!Parent::Func(void)}	void *
[1]	0x00007ff689851721 { 연습용.exe!Parent::FF(void)}	void *
a	-858993460	int
c	{...}	Child
Parent	{a=-858993460 }	Parent
__vfptr	0x00007ff689864510 { 연습용.exe!void(* Child::`vftabl...	void **
[0]	0x00007ff689851730 { 연습용.exe!Child::Func(void)}	void *
[1]	0x00007ff689851721 { 연습용.exe!Parent::FF(void)}	void *
a	-858993460	int

자식 클래스의 가상함수 테이블에서는 자식 클래스에서 오버라이드 된 함수를 가리키는 것을 볼 수 있다.

1.1.8 C++ 스타일 캐스팅

단순히 (int*) 와 같이 캐스팅하는 C스타일과 달리, C++은 4개의 캐스팅 방법으로 나뉘어진다.

const_cast

해당 포인터 객체의 상수성을 제거한다.

```
const int* const_;  
int* not_const = const_cast<int*>(const_);  
  
*const_ = 4;  
^  
*not_const = 4;
```

상수성은 항상 유지되어야 하지만, 외부의 다른 라이브러리를 사용할 때 인자값으로 비상수성 객체를 요구한다면 하는 불가피한 상황에서 한시적으로 사용한다.

단, `const_cast`를 사용하는 것은 해당 객체를 인자로 받는 함수 내에서 해당 상수변수를 수정하지 않는다는 것을 알고 있을때만 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 생각해 보아야 한다.

static_cast

일반적인 캐스팅 방식

static_cast의 단점은, 런타임 타입 검사를 하지 않는다는 것이다.

static_cast는 업캐스팅과 다운캐스팅을 둘 다 할 수 있는데,

계층 구조가 실제로 유효한지는 검사하지 않는다.

즉 자식 클래스를 부모 클래스로 업캐스팅하여 사용하다가(다형성을 위해),

다시 자식 클래스로 다운캐스팅하는 것은 유효하며 정상적인 사용법이지만,

순수한 부모 클래스인 객체를 자식으로 다운캐스팅하여 사용하는 것도 막지 않는다는 것이다.

이런 식으로 캐스팅하게되면 메모리 침범 등 심각한 오류가 발생한다.

계층 구조 간 안전한 캐스팅을 하려면 **dynamic_cast**를 사용한다.

대신 **dynamic cast**는 **virtual** 함수가 하나라도 있어야 함. **vtable**에 **RTTI**가 저장되기 때문에.

RTTI : RunTime Type Information

dynamic_cast

런타임 타입 검사를 수행하는 안전한 다운캐스팅에 사용하는 캐스팅 방법이다.

다이나믹 캐스트는 타입이 유효한지 체크하기 위해 런타임에 타입 검사를 수행하기 때문에, 런타임 비용이 높아서 잘 사용되지 않는다. 사실 쓰는것도 못봤다.

reinterpret_cast

타입을 강제 변환한다. **double**을 **byte**로 변환하는 등의 임의 변환이 가능하다.

타입을 비트 단위로 1:1 변환시킨다. 그렇기에 원본 데이터가 소실되거나(데이터가 찢리거나), 원하지 않는 결과가 나올 수 있으므로 주의해야 한다.

1.1.9 STL 컨테이너

벡터(Vector)

동적 배열(크기조절 가능), 배열과 같이 연속된 자료구조이므로 캐시 친화적이다. 임의접근반복자(**random access iterator**)를 사용하므로 배열의 원소에 즉시 접근 가능하다.

(반복자를 5가지를 알아두면 도움이 된다)

emplace를 하면 복제 과정 없이 바로 원소를 삽입할 수 있어서 **push_back**보다 비용이 절감된다.

반복자 무효화가 발생할 수 있다.

반복자 무효화

컨테이너의 메모리가 재할당 될 때(**vector resize**, **push_back** 등)나 요소 삭제를 하는 등의 동작에서 발생할 수 있습니다.

만약 STL의 **erase**를 사용한다면 **erase**한 원소 포함해서 뒤의 원소를 가리키는 모든 반복자가 무효화됩니다. 이는 삭제 후 뒤에 있는 요소를 모두 뺏겨줘야 하기 때문입니다.

단, **erase** 함수는 다음주소를 가리키는 **iterator**를 리턴해주기 때문에, 이것을 사용하여 순회를 계속 할 수 있습니다. 그러므로, **for**문과 같이 **iter**를 계속해서 ++해주는 코드를 사용할 때 주의해야 합니다.

벡터 요소의 삽입

삽입한 위치 뒤를 가리키는 반복자들은 무효화된(나머지 원소를 전부 밀어줘야하기때문에), 삽입 앞은 재할당에 따라 무효화될수도있고 아닐수도있으나 안전한 프로그래밍을 위해선 무효화라고 보는게 맞다(삽입 앞은 원소의 위치 변동이 없으나, 메모리 재할당(크기 초과로 인한 재할당 등))이 일어날 수 있기 때문이다.

벡터 요소의 삭제

삭제 뒤는 무효화(뒤에 있는 원소들을 뺏겨줘야하니까), 삭제 앞은 무효화되지 않음
(재할당이 일어날 일이 없다)

리스트(List)

단/양방향 반복자를 사용한다. 이로 인해 임의 접근이 불가능하므로, 어떤 원소에 접근하기 위해선 해당 위치까지 순회해야 한다.(list의 **advance** 함수는 임의 접근처럼 보이지만 실 구현은 순회로 구현되어 있다) 삽입과 삭제가 상수 시간을 가진다. 포인터를 사용하여 서로를 연결. 삽입과 삭제가 빈번한 구조에 사용하기 좋다.

맵(Map)

균형이진트리의 일종인 레드-블랙 트리로 구현되어 있다. 키-값으로 저장하고 키로 검색하여 값을 찾는다. 이진 탐색을 사용하므로 검색이 빠른 편이다. 삽입, 삭제 시 균형을트리이기 때문에 균형을 유지하는데에 오버헤드가 생긴다.

맵에 삽입하는 경우 이미 있는 **key**라면 값을 삽입하지 않고 이미 키가 있는 해당 노드의 키/값을 리턴한다.

정확히는 map은 insert를 할 경우 무조건 **pair**를 리턴하는데

pair< std::pair(key,value) , bool > 을 리턴한다. **bool**은 해당 **key**가 이미 맵에 있느냐, 없느냐를 의미한다. 해당 키가 있으면 **false**, 없으면 **true**리턴. 앞의 **pair**는 키,값이다.

unordered_map : 해시맵. 검색이 빠르다(해시 함수로 인한 상수 시간 탐색 가능). 여유 공간이 실제 공간보다 많이 필요하다. 공간을 적게 잡으면 충돌이 자주 일어난다.

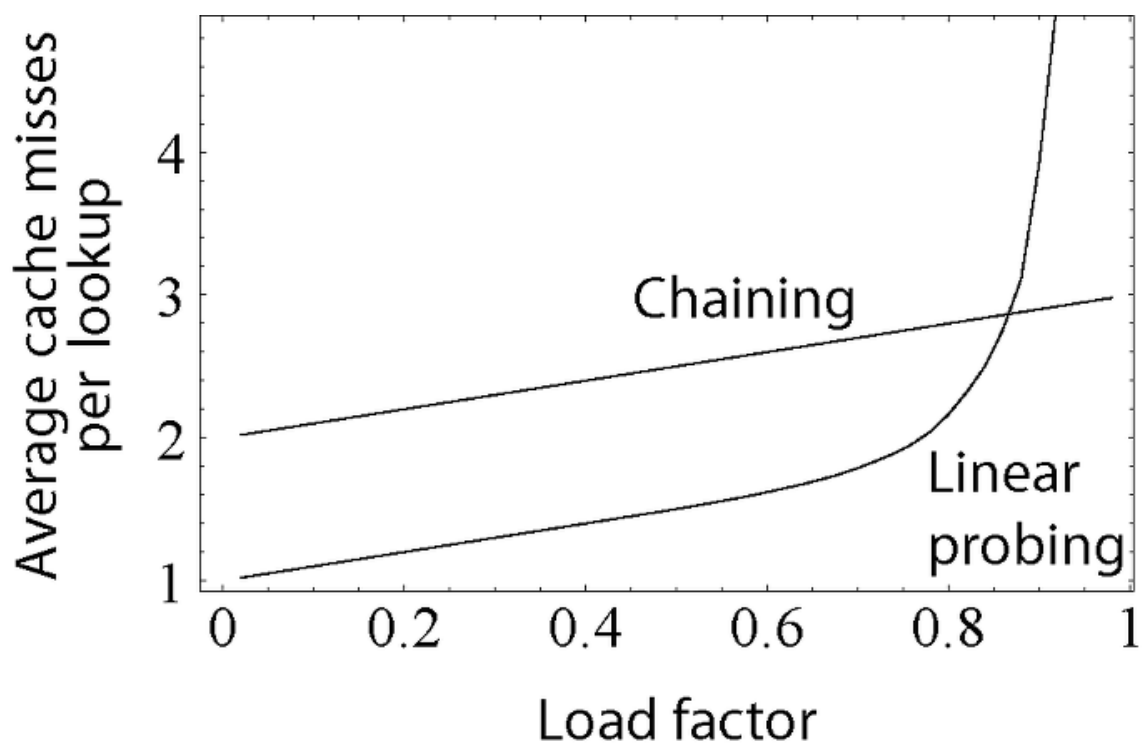
해시맵을 구현하는 방법으로는 개방 주소법, 체이닝이 있다.

해시맵의 구현법 별 장점

체이닝(Chaining)의 장점

→ 연결 리스트만 사용하면 된다. 즉, 복잡한 계산논리를 사용할 필요가 개방주소법에 비해 적다.

→ 해시테이블이 채워질수록, **Lookup** 성능저하가 선형적으로 발생한다. (그림 참조)



개방 주소법(**Open Addressing**)의 장점

◎ 개방주소법(**Open Addressing**)의 장점

- 체이닝처럼 포인터가 필요없고, 지정한 메모리 외 추가적인 저장공간도 필요없다.
- 삽입, 삭제시 오버헤드가 적다.
- 저장할 데이터가 적을 때 더 유리하다.

개방 주소법의 탐색 종류

선형 탐색(선형 조사)(**Linear Probing**): 해시충돌 시 다음 버킷, 혹은 몇 개를 건너뛰어 데이터를 삽입한다.

제곱 탐색(**Quadratic Probing**): 해시충돌 시 제곱만큼 건너뛴 버킷에 데이터를 삽입(1,4,9,16..)

이중 해시(**Double Hashing**): 해시충돌 시 다른 해시함수를 한 번 더 적용한 결과를 이용함

1.1.10 가비지 컬렉터

C++의 스마트 포인터는 레퍼런스 카운팅 방식을 사용(가비지 컬렉션이 아님), **C#**이나 언리얼은 나이트마크 앤드 스왑 방식을 사용.

레퍼런스 카운팅 : **C++** 스마트 포인터에 사용됨. 객체마다 레퍼런스 카운트를 두어 객체의 참조개수를 카운트한다. 카운트가 0이 되면(해당 객체를 사용/참조하는 부분이 없으면) 해당 객체를 해제한다.

장점 : 대개 메모리를 사용 직후 해제하므로 해당 객체가 캐시에 있을 확률이 높으며, 그로 인해 해제가 빠르게 이루어진다.

단점 : 레퍼런스가 0이 될 때 참조한 객체가 많으면 시간이 오래 걸릴 수 있으며, 멀티스레드 환경에서 카운팅에 대한 데이터레이스가 일어날 수 있다. 순환참조가 생길 수 있다.

나이프 마크 앤 스위프 : 모든 객체의 리스트를 루트 셋에서 가지고 있다. 해제할 메모리는 루트 셋에서 제외한다. 해제하지 않아야 할 메모리는 가지고 있다. 이게 **Mark** 단계. **Sweep** 단계에선 연결되지 않은 메모리를 싹다 해제한다.

예전에 스터디에서 해당 내용에 대한 주제가 나온 적이 있었는데.. 그때 들은 바로는 언리얼 GC 객체가 생성되는 모든 **UObject**들을 **Object** 어레이에 따로 기록을 해 두고, **Flag**값을 조절하여 해당 메모리를 해제할지 여부를 알아낸다. 그러므로 언리얼에서 GC로 관리할 수 있는 객체는 **Uobject**를 상속한 객체 뿐이다.

또한 **UPROPERTY**, **UCLASS** 등으로 리플렉션 데이터를 만들어야 GC를 쓸 수 있다.

단점 : 메모리 단편화가 발생할 수 있으며, 작업 도중 메모리가 변경되면 안되기 때문에 시스템이 중단되어야 한다. (실시간성이 중요한 게임에선 심각한 문제가 될 수 있다)

1.1.11 타입 간의 계산 결과

```
int b = 5, c = 4, d = 2;
```

```
float a = d + b/c;
```

b/c 는 int끼리의 계산이므로, $5/4 = 1$ 이다.

즉 답은 3.0

즉 변수의 리턴타입은 중요하지 않고, 어떤 타입끼리 계산을 하는지가 중요하다.

$\text{float} = \text{float} / \text{int}$ 일 경우, float/int 에서 **float**로 계산된다. 즉 더 넓은 표현 타입으로 계산되는 것.

$\text{int} = \text{float} / \text{int}$ 일 경우, float/int 로 계산되지만 리턴타입인 **int**로 잘려서 들어간다.

정리하자면 , 최종 결과가 계산되기 전 까지의 값은 계산한 타입끼리 임시변수로 만들어진다고 생각하면 편할 것 같다.

int = int / float + int / int + float / float; 와 같은 식을 풀이하면

int = int / float □ 임시변수 : float int / int □ 임시변수 : int

float / float □ 임시변수 : float;

최종 결과 : int (float + int + float를 더한 값에 최종 변수가 int이므로 소수점 버림)

1.1.12 생성자와 소멸자

사례 1.

```
class B
{
public:
    B() { std::cout << "constructor" << std::endl; }
    ~B() { std::cout << "destructor" << std::endl; }
    B(const B& b) { std::cout << "copy constructor" << std::endl; }
    B(B&& b) { std::cout << "move constructor" << std::endl; }
};

B Func()
{
    B b;
    return b;
}

int main()
{
    B d = Func();

    return 0;
}
```

결과

```
Microsoft Visual Studio
constructor
move constructor
destructor
destructor
```

왜?

1. Func안의 B b; 생성으로 인해 기본 생성자
2. b를 리턴하는데, 값 복사가 아닌 이동으로 처리됨(NRVO)
3. move를 통해 꺾이기만 남은 **b** 소멸자 호출
4. main()함수의 return 0을 통한 소멸자 호출
5. 만약 Release모드라면, 모든 과정에서 constructor만 호출된다.(최적화)

사례 2.

```
class T
{
public:
    T() { cout << "constructor" << endl; a = 4; }
    ~T() { cout << "destructor" << endl; a = 2; }
    T(const T& t) { cout << "copy constructor " << endl; }
    T& operator=(const T& t) { cout << "대입 연산자" << endl; }
    T(T&& t) noexcept { cout << "move constructor" << endl; a = t.a; }
    T& operator=(T&& t) { cout << "이동 대입 연산자 " << endl; }

    int a;
};
```

```

T Func4(T t)
{
    T d = t;
    return d;
}

void Func3(T t)
{
    Func4(std::move(t));
}

T Func1()
{
    T t;
    return t;
}

void Func2(T t)
{
    Func3(t);
}

int main()
{
    Func1();

    Func2(Func1());
}

```

constructor
move constructor
destructor
destructor
constructor
move constructor
destructor
copy constructor
move constructor
copy constructor
move constructor
destructor
destructor
destructor
destructor

왜?

1. Func1()내의 T t; 생성으로 인한 기본 생성자 호출
2. d를 리턴하는데, NRVO에 의해 이동 생성자가 호출
3. 겹데기만 남은 t가 소멸.
4. main()함수에서 이동생성자를 통해 이동된 T t가, 리턴할 곳이 없음으로 소멸
5. Func2()의 인자인 Func1의 T t; 가 생성되어 기본 생성자 호출

6. 마찬가지로 NRVO에 의해 리턴 시 이동생성자 호출
7. 리턴 후 꺾데기만 남은 t가 소멸됨. 소멸자 호출
8. 이동생성자를 통해 Func1에서 리턴된 t가, Func2의 인자로 들어간다. Func1의 리턴값은 우측값이기 때문에, 복사 없이 들어가므로 아무 생성자도 호출되지 않는다.
9. Func3()의 인자로 t를 넣기 위해 복사생성자가 호출된다.
10. Func4는, Func3의 std::move를 통해 이동된 우측값이므로, 이동생성자가 호출된다.(move는 이동이 가능하다는 것을 명시하여 컴파일러가 이동생성자를 호출하게 하기 위한 도구이므로, 이동생성자가 호출되어야 한다.)
11. Func4의 T d = t;는 T d를 최초 생성하는 부분이므로, 대입연산자가 아니라 복사생성자가 호출되게 되므로, 복사생성자가 호출된다.
12. return d;(NRVO)를 통해 이동생성자가 호출된다.
13. Func4의 인자값인 T t가 소멸.
14. Func4에 이동하고 남은 꺾데기가 소멸
15. Func4의 리턴값은 갈데가 없으므로 소멸자가 호출
16. Func3의 t 꺾데기가 소멸
17. Func2의 Func1에서 이동된 t가 소멸

RVO, NRVO(Return Value Optimization, Named Return Value optimization)

임시객체가 리턴되면, 요즘 컴파일러는 해당 리턴을 값복사로 처리하지 않고 이동으로 처리하게 된다. 이를 RVO라고 한다.

원래 T t; return t;와 같이, 이름이 있는(주소와 메모리가 있는, 임시객체가 아닌) 객체들은 RVO가 적용되지 않았으나, C++ 개정시에 같이 적용되게 되어 NRVO로 불리게 되었다.

1.1.13 C++의 메모리 영역

스택 메모리 영역

클래스의 멤버변수와 지역변수가 할당되며, 함수의 파라미터, 함수의 반환 주소값이 들어감

스택 자료구조처럼 쌓이기 때문에, 함수가 끝나면 하나씩 pop해가면서 반환주소를 찾는다

특징은 선언 범위를 벗어나면(Scope를 벗어나면) 자동으로 해제되기 때문에 따로 메모리 해제를 신경 쓸 필요가 없다.

스레드는 이 영역을 공유하지 않는다. 독립적으로 가지고 있음

데이터 영역

정적(**static**), 전역변수가 저장되는 공간

전역 변수의 경우 프로그램의 시작과 함께 할당되며, 할당 순서가 미정이기 때문에, 선언 및 사용 시 타이밍에 주의하여야 하며 해제 순서는 할당의 역순입니다.

코드 영역

실행될 코드가 저장(읽어야 하는 명령어도 메모리에 존재해야 한다)

힙 영역

메모리 공간을 동적으로 할당/해제 할 수 있는 영역

스레드는 이 영역을 공유합니다.

1.1.14 C++11/14에 추가된 신기능

범위기반 for문(range based for)

람다식(lambda)

자료형 추론(auto)

enum class (scoped enum)

override, final 키워드

진보된 난수생성기(랜덤숫자 생성기)

mt19937 : mt19937은 메르센 소수의 주기를 이용한 의사난수 생성기이다.

빠르고, 난수 주기가 크지만 암호학적으로 비교적 안전하지 않다. 그렇기 때문에 **mt19937**의 시드 넘버를 생성하는데에 **random_device**를 이용한다.

random_device : 하드웨어 리소스와 같은 예측하기 어려운 값을 통해(CPU 클럭값 같은 것을 이용함) 비결정적 난수를 생성한다.

스마트 포인터

해시맵

등등....

1.1.15 const

```
char greeting[] = "Hello";
```

char *p = greeting; //포인터가 가리키는 주소를 바꿀수도 있고, 가리키는 값의 내용 바꿀수도 있다.
(비상수 포인터, 비상수 데이터)

const char *p = greeting; //포인터가 가리키는 주소를 바꿀 수 있지만 가리키는 값의 내용은 바꿀 수는 없다(비상수 포인터, 상수 데이터)

char * const p = greeting; //포인터가 가리키는 주소를 바꿀 수 없고, 가리키는 값의 내용을 바꿀 수 있다.(상수 포인터, 비상수 데이터)

const char * const p = greeting; //포인터가 가리키는 주소를 바꿀 수 없고, 가리키는 값의 내용 또한 바꿀 수 없다(상수 포인터, 상수 데이터)

const 키워드가 *의 왼쪽에 있으면 포인터가 가리키는 주소를 바꿀 수 없고, *의 오른쪽에 있으면 값의 내용을 바꿀 수 없다.

1.2 컴퓨터공학

여기서부터


<https://www.slideshare.net/soochanpark/cs-234864069>


와 같이 보는 것이 좋다.

1.2.1 프로세스와 스레드

프로세스

프로세스는 '실행 중인 프로그램'으로 정의한다. 즉 실행되고 있는 각각의 프로그램이 (심지어 같은 프로그램을 여러 개 실행시켰더라도) 모두 프로세스이다.

>  Microsoft Visual Studio 2017(32비트)(2)

>  Microsoft Visual Studio 2017(32비트)(9)

작업 관리자 – 프로세스 창에서 본 2개의 비주얼 스튜디오.

똑같은 비주얼 스튜디오 프로그램이지만, 각각은 다른 프로세스이다.

프로세스들은 각자 고유한 가상 메모리를 가지고 있기 때문에, 실제 물리 메모리 사용량과는 상관없이 프로세스 자신이 전체 메모리를 전부 가진 것처럼 작동하며, 이러한 착각을 지원하기 위해 가상 메모리 할당과 페이징 같은 기능들을 제공한다.

그렇기 때문에 다른 프로세스와 통신하려면 (IPC : Interprocess Communication) 메시지 패싱이나 공유 메모리(Shared Memory) 같은 방식을 사용해야 하며, 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 가지게 된다.

컨텍스트 스위칭이 스레드보다 무겁다(스위칭에 필요한 정보들이 스레드보다 많기 때문이다. 단 대개 유의미할 정도로 문제가 되진 않는다)

스레드

스레드는 각각의 프로세스의 실행 흐름이다. 하나의 프로세스에서 여러 개 스레드를 만들 수 있다. 레지스터와 스택 메모리를 제외한 힙, 데이터 공간을 공유한다. 이로 인해 빠른 스레드 간 통신을 할 수 있으며 컨텍스트 스위칭이 가볍고 빠르다. 스레드의 컨텍스트 스위칭 기법은 FCFS(First Come First Served), RR(Round-Robin : 시분할), SJF(Shortest Job First) 등의 기법이 있다.

메모리를 공유하기때문에 그에 따른 Data Race, Deadlock 문제를 피할 수 없어서 Lock이나 Atomic을 사용하여 잘 회피하여야 한다. 스레드는 시간이 일정 이상 경과하거나(Timeout), IO 인터럽트가 들어오거나, Sleep 등의 코드가 실행되거나, 스레드 자신이 CPU 점유를 양보하거나(Yield) 등의 이유로 스위칭 될 수 있다.

1.2.2 데드락

락(Lock)

락이란 락을 걸어놓고 들어간 해당 스레드를 제외한 어떤 스레드도 락이 걸린 구역으로 들어가지 못하는 것을 의미한다. 락을 걸어놓은 해당 스레드가 락을 릴리즈(해제) 하고 해당 구역에서 나가면, 다른 스레드가 접근할 수 있게 된다.

앞으로 데드락 설명에서는 자원 == 코드 구역 이라고 봐도 된다. 크리티컬 섹션 안의 명령어들도 스레드가 독점해야 할 자원으로 취급하기 때문이다.

즉 그 해당 스레드에서 다른 스레드로 context switching이 될 수는 있지만, lock이 된 구역은 해당 스레드 외엔 누구도 진입하지 못하게 됨을 의미한다.

메모리를 공유하여 읽기/쓰기를 수행할 스레드들은 공유 메모리에 접근하기 전에 메모리에 대해 락을 시도해야 한다. 읽는 중에 데이터가 바뀌거나 하는 사태를 방지하기 위함이다. 다른 스레드가 이미 점유중일 경우, 새로 락을 시도하려던 스레드는 점유중인 스레드가 락을 해제하거나, 타임아웃이 될 때까지 대기하게 된다.

데드락(Deadlock)

만약 A,B자원을 둘 다 사용해야 하는 스레드 C,D가 있는데 스레드 C가 A를 , 스레드 D가 B를 가지고 서로의 객체를 반납하기만을 기다리고 있다면 이러한 상태를 데드락이라 한다.

즉 데드락이란, 특정한 자원(메모리, 레지스터, CPU 연산장치 등을 의미)을 사용하는 스레드들이 서로의 자원을 반납하기만을 무한히 기다린다면, 이것을 데드락이라고 한다.

유명한 예시로는 식사하는 철학자 문제가 있다.

데드락은 멀티스레드 환경에서는 매우 간단하게 발생할 수 있으며, 발생하면 해결하기 어려운 문제이다.

락을 구현하기 위한 방법들

세마포어(Semaphore)

두 개의 Atomic operation인 wait(P)와 signal(V)로만 접근이 가능한 방법이다.

P = 세마포어 카운트를 1 깎고 연산에 진입한다. (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 Wating 방지)

세마포어는 0이하로 떨어질 수 없다.

P와 V가 Atomic(원자적)하기 때문에, P->P, V->P 등의 동작을 하게 될 경우 데드락이 발생하거나, 상호배제(각 스레드가 자원에 대해 서로를 배타적으로 배제함)를 보장할 수 없게 되는 약점이 있다. 또한

A. P(s) ->P(q) | Critical Section | V(s)-> V(q)

B. P(q) -> P(s) | Critical Section | V(q) -> V(s)

와 같이 설계된 세마포어 코드가 있을 때, A의 P(s)와 B의 P(q)가 같이 일어난다면 서로가 s와 q를 한 손에 든 상태에서 상대의 자원이 V되기를 기다리게 되므로 데드락이 발생한다.

(A는 P(q)를 위해 B가 q를 반납하길 기다릴거고, B는 P(s)를 위해 A가 s를 반납하길 기다릴 것이므로...)

해결법으로는 세마포어가 모든 자원을 다 얻은 채로 프로세스를 진행하게 하거나(다 얻지 못하면 그 즉시 모든 자원을 반납하고 재시도한다), 타임아웃을 넣거나 하는 방법이 있다.

Mutual Exclusion(Mutex)

0,1만을 사용하는 Binary Semaphore를 Mutual Exclusion(mutex)라고 한다.

상호 배제를 구현할 수 있다. (세마포어 카운트가 최대 1이면 자원에 한 번에 하나의 스레드만 접근할 수 있기 때문이다)

C++에서 스레드/락의 구현 방법으로는 `_beginthreadex`와 `waitforsingleobject`등을 쓰는 윈도우 스레드와 STL에서 추가된 `std::thread`를 쓰는 방법이 있다. 스레드 같은 경우는 ‘실제로 구현해 보았는지?’를 물어보는 질문이 가끔 등장하므로 실제로 스레드를 이용한 프로그램을 구현해 보는 경험을 해 보는 것이 필요할 것 같다.

1.2.3 메모리

<https://www.slideshare.net/soochanpark/cs-234864069> 참조

우리가 말하는 ‘메모리’는 대개 주 메모리(Main memory)인 RAM 공간을 의미하는데, 이는 ‘물리 메모리’라고 한다. 프로그램은 자기가 사용할 데이터를 이 물리 메모리에 미리 가져와서 필요할 때 사용한다.(하드디스크는 굉장히 느린 매체이므로, 필요할 때 하드디스크에서 데이터를 로드한다면 컴퓨터를 쓸 수 없을 수준이 된다)

또한 데이터가 더 이상 필요하지 않다면, 메모리에서 추방하여 다른 데이터가 들어올 수 있게 만든다.

여기서부터는 메인 메모리 = 물리 메모리 = 주 메모리 = **RAM** 이다.

가상 메모리

가상 메모리는 가상의 메모리 공간이다. 가상의 메모리 체계를 만들어서, 실제로 사용되는 물리 메모리 공간에 1:1 대응시킨다.

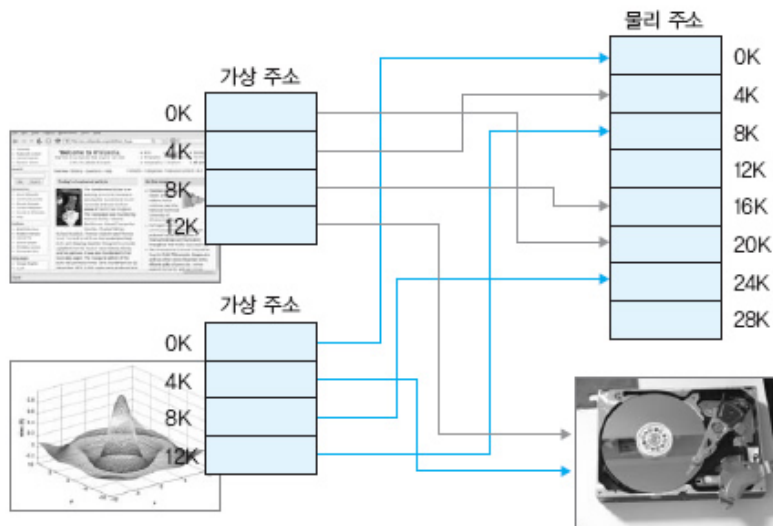


그림 3-3 가상 주소를 물리 주소로 사상(mapping)

RAM(물리 메모리)가 있는데 굳이 가상으로 메모리 공간을 만드는 이유는 여러 가지가 있다,

첫 번째는 각각의 프로세스가 독립적인 가상 메모리 공간을 가지게 하기 위함이다.

각 프로세스마다 연속적이고 충분히 큰 가상메모리를 만듦으로써, 실제 물리 메모리가 어떨든 상관없이 더 큰 메모리를 사용할 수 있게 하며, 프로세스가 자신이 메모리 자원을 전부 점유하고있다고 착각할 수 있게 만든다.

즉, 내가 만약 **16GB**의 물리 메모리 공간을 가지고 있다고 하고, **5개**의 프로세스가 이를 나눠 쓴다고 하자.

만약 가상 메모리 공간을 **16GB**로 잡아 놓으면, 물리 메모리가 얼마나 남아있던 간에 각각의 프로세스는 **16GB**의 공간을 모두 자유롭게 사용할 수 있다. (물론 상식적으로 말이 안 되기 때문에, 물리 메모리의 용량이 깎 차면 운영체제는 안 쓰는 다른 프로세스의 데이터를 물리 메모리에서 몰래 내쫓는 등의 공간 확보 작업을 한다)

정확히는 모두 자유롭게 사용할 수 있다고 착각하게 만든다.

이는 프로그래머가 물리 메모리의 사용량이나 점유현황을 신경쓰지 않고도 구현이 가능하게 만드는 이점도 있다.

또한, 가상 메모리는 프로세스마다 독립적이므로 프로세스가 각자의 데이터를 함부로 접근하거나 조작할 수 없어서 운영체제의 메모리 안정성을 높인다.

90년대에는 이러한 가상 메모리가 완벽하지 않아서, 프로그래머가 메모리 공간을 직접 잡아서 부여하는 작업을 해야 하기도 하고, ppt 작업 중 mp3 플레이어를 실행시켰더니 ppt가 터지는 일도 발생하였다고 한다.

두 번째는 외부 단편화를 완화하는 효과가 있기 때문이다.

단편화(Fragmentation)

외부 단편화



좌측이 전체 메모리, 우측이 새로 할당해야 할 메모리이다. 흰색 부분이 빈 메모리이다.

그림과 같이, 빈 물리 메모리의 전체 공간이 충분(35)함에도, 메모리가 연속적이지 않아서 메모리를 새로 할당(21) 할 수 없다. 이를 '외부 단편화' 라고 한다.

내부 단편화



그림과 같이 메모리를 할당할 때, 4만큼 블록 단위로 할당하였다. 만약 메모리가 7만큼 할당이 필요하다면, 크기 4의 블록 2개를 할당해주어야 하고, 1만큼의 메모리가 낭비되게 된다. 이를 '내부 단편화'라고 한다.

단편화의 해결법

A. 메모리 풀

메모리 풀(memory pool)은 고정된 크기의 블록을 할당하여 [malloc](#)이나 [C++의 new 연산자](#)와 유사한 [메모리 동적 할당](#)을 가능하게 해준다. [malloc](#)이나 [new 연산자](#) 같은 기능들은 다양한 블록사이즈 때문에 [단편화](#)를 유발시키고, 파편화된 메모리들은 퍼포먼스 때문에 [실시간 시스템](#)에서 사용할 수 없게 된다. 좀더 효율적인 방법은 **memory pool**이라고 불리는 동일한 사이즈의 메모리 블록들을 미리 할당해 놓는 것이다. 그러면 응용 프로그램들은 [실행 시간](#)에 [핸들](#)에 의해서 표현되는 블록들을 할당하고, 접근하고, 해제할 수 있다. (위키백과)

프로그래머가 할 수 있는 외부, 내부 단편화의 해결법이다. 프로그래머가 직접 메모리를 관리할 수 있는 특정 크기의 풀(pool)을 만들어서 그 풀에서 메모리를 가져오고, 사용이 끝나면 돌려주는 것이다. 미리 만든 풀에서 메모리를 가져오므로 할당이 새로 일어나지 않는다.

장점

단편화를 완화할 수 있다. 할당/해제가 빈번할 때 **new/delete**를 통해 새로 할당/해제할 필요가 없이 풀에서 가져오고, 반납하면 되므로 비용이 크게 감소한다.

단점

메모리 풀은 프로그래머가 직접 메모리를 0부터 100까지 관리하겠다고 선언하는 것과 다름이 없다. 즉 모든 메모리 문제의 책임은 프로그래머가 가지게 된다.

또한 사용하지 않을 때도 풀을 유지해야 하므로 메모리 낭비가 생긴다.

B. 페이징

페이징이란 가상 메모리를 페이지로 나누어 다루는 기법이다.

페이지(가상 메모리 나눴의 단위)와 프레임(물리 메모리 나눴의 단위)은 1:1 대응되며 (가상 메모리의 주소값과 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 외부단편화를 완화한다.

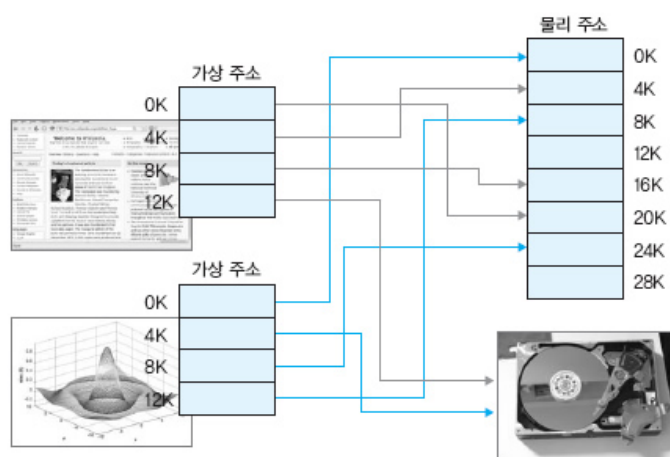
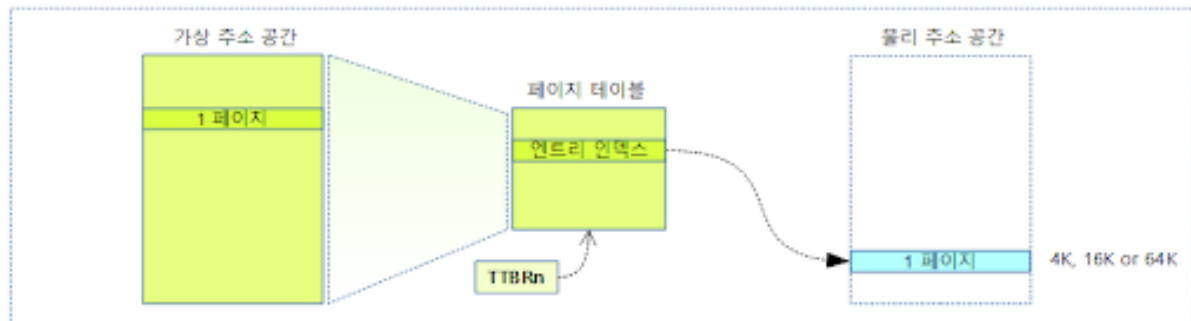


그림 3-3 가상 주소를 물리 주소로 사상(mapping)

그림과 같이 물리적으로 연속적이지 않아도 가상 주소는 연속적으로 만들 수 있다.

1:1 사상관계(주소값 정보)를 ‘페이지 테이블’에 기록하여, 페이지를 통해 프레임을 찾아서 원하는 데이터를 얻을 수 있게 만든다. 페이지 테이블은 메인 메모리에 저장된다.



즉 가상 메모리 주소를 근거로 페이지 테이블에서 물리 메모리 주소를 찾아서, 물리 메모리 데이터에 접근하는 것이다. (실제 데이터는 물리 메모리에 들어 있으므로)

페이지 폴트

물리 메모리 안에서 페이지에 대응되는 프레임이 실제로 존재하지 않을 때 발생한다. 왜냐하면 가상메모리를 통해 각각의 프로세서들은 전체 메모리를 혼자 점유한다고 착각하고 사용하고, 실제 메모리는 이러한 착각을 지원하기 위해 물리 메모리가 가득 차면 사용하지 않을 것 같은 데이터를 메모리에서 내쫓고 새로운 데이터를 들여오기 때문에, 내가 필요한 페이지가 정작 필요할 때 없을 수 있기 때문이다.

페이지 폴트는 디스크를 이용한 데이터 로드가 필요하기 때문에 매우 성능적 비용이 크게 발생하는 문제이므로 최대한 피할 수 있도록 해야 한다.

쉽게 말해, 메모리가 부족하면 생기는 현상이다.

디맨드 페이징

페이지 폴트가 발생하면, 결국 디스크에서 데이터를 가져와야 한다.

이를 요구 페이징(**demand paging**)이라고 하며, 매우 느린 저장장치인 2차 저장장치(하드디스크)에서 가져오는 것이니만큼 많은 시간 손실이 발생한다. 이래서 프로그래머는 메모리를 다룰 때 페이지 폴트를 최대한 줄이는 방향으로 구현해야 한다.

1.2.4 네트워크

TCP/UDP

1. TCP : 연결지향형 서비스(reliable service)와 신뢰적 데이터 전송 서비스(data integrity)를 포함한다.

TCP의 특징 -> 이 용어를 외울 필요는 없음. 내용은 외울 필요 있음

reliable service

TCP는 메시지 전송 전에 클라이언트와 서버가 서로 전송 제어 정보를 교환하도록 한다. 즉, 패킷이 전송될것을 미리 알려준다.

data integrity

TCP는 데이터를 오류 없이 올바른 순서로 전달하는 것을 보장한다.

congestion control(혼잡제어)

TCP의 congestion control은 네트워크가 혼잡상태에 이르면 프로세스의 속도를 낮춘다. 즉, 프로세스가 우선이 아니라 네트워크의 안정성 향상을 우선한다.

flow control(흐름제어)

TCP는 receiver의 buffer가 꽉 차서 데이터가 손실되지 않도록 sender의 전송을 관리한다.

요약하자면 TCP는 최대한 오류 없는 신뢰성 있는 전송을 위해 많은 기능을 지원하며, 이에 따라 속도가 UDP에 비해 느릴 수 있지만 안정적이고 정확한 패킷 전달을 보장한다는 특징이 있다.

TCP의 사용처

HTTP, 신뢰성 있는 데이터 전송이 필요한 대부분의 서비스에 사용된다. 게임도 대개 TCP를 사용한다.

2. UDP : 최소의 서비스 모델을 가진 간단한 프로토콜이다. 비연결형임으로 핸드셰이킹을 하지 않는다.

비신뢰적인 서비스이므로 데이터가 손실될 수 있다. UDP는 flow control을 하지 않으므로 프로세스는 원하는 속도로 하위 계층으로 보낼 수 있다. 그 외 올바른 패킷 전송에 필요한 여러가지 제어를 전부 하지 않는다.

데이터그램을 생성하며, 데이터 전송의 신뢰성을 보장하지 않는다. 비연결형이고, 데이터를 보내고 신경쓰지 않는다. 재전송 또한 없다.

그래서 **TCP**보다 비교적 빠르다. 연결 상태를 만들지 않으므로 유지할 필요도 없다.

또한 패킷이 순서대로 오는 것을 보장하지 않기 때문에 수신자가 헤더를 보고 조합해야 한다.

최소한의 오류검출은 한다. 그렇다고 오류가 있으니 다시 보내달라고 하진 않는다

요약하자면 **TCP**보단 빠르지만 안전하지 못하다. 패킷 유실이 가능하다.

UDP의 사용처

DNS, 패킷 에러에 민감하지 않은 음성 통화, 비디오 서비스, 화상 통화 등..

(유튜브 영상의 픽셀 하나가 잘못 전송되었다고 해도 아무도 눈치채지 못한다. 이런곳엔 **UDP**가 적합하다. 실제로 유튜브가 **UDP**를 쓰는지는 나도모르겠음.)

Nagle 알고리즘

TCP 네트워크에서, 데이터는 **OSI**레이어를 거치면서 몇 겹의 헤더로 캡슐화되어 목적지로 보내진다. 또한, 패킷 전송에는 비용이 따른다. 이런 경우 데이터가 적다면 보내는 효율이 떨어지게 된다. 적은 데이터를 가진 패킷을 여러번 보내는 것보다는 많은 데이터를 가진 패킷을 한번에 보내는 것이 효율이 좋다. (네트워크 부하가 적어진다) 이럴 경우, 상대가 받을 수 있는 사이즈(**window size**)가 충분하다면 크기가 작은 패킷을 모아서 보내는 것을 **Nagle** 알고리즘이라고 한다. 이렇게 보내면 전송의 효율이 증가하지만, 즉각적인 반응이 필요한 네트워크 게임 등에는 적합하지 않다(점프 입력을 세번 해야 패킷이 간다고 생각하면...?) 즉 네트워크 게임에서는 네이글 알고리즘을 **off**해야 한다.

네이글 알고리즘은 패킷전송제어에 관한 구현이므로, **UDP**에서는 사용할 수 없다.

1.3 컴퓨터 그래픽스

대개 포트폴리오에 **DirectX**나 그래픽스 관련 내용이 있으면 물어봅니다. 자기 포폴에 맞춰 준비하되(포폴에서 **Picking**을 구현하였다면 피킹에 대한 내용을 공부해 간다면), 렌더링 파이프라인의

개략적인 구조와 정의 같은건 범용적으로 외워 가면 좋습니다.

1.3.1 렌더링 파이프라인

렌더링 파이프라인이란, 3차원의 기하 정보 데이터를 최종적인 모니터에 출력되는 2차원 이미지로 바꾸어주는 일련의 단계들을 의미함

즉 처음에 버텍스와 인덱스로 저장되어있는 모델의 데이터를, 모니터에 띄울 수 있는 형태로 바꿔주는 과정을 의미한다.

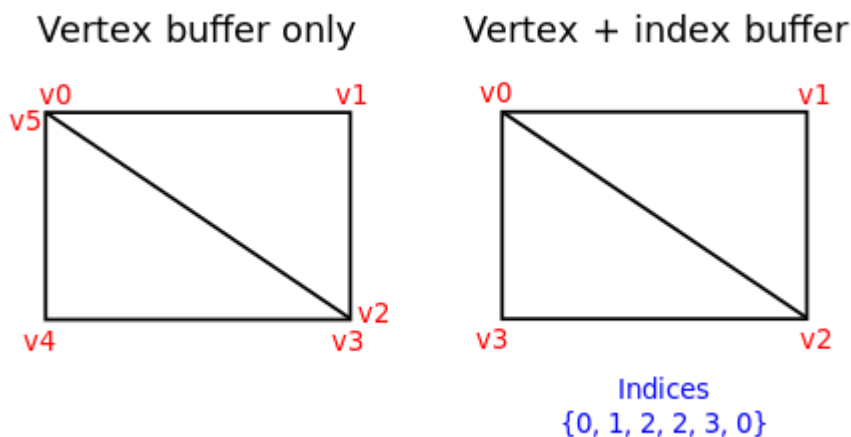
Input Assembler (IA, 입력 조립기)

전송받은 버텍스 버퍼와 인덱스 버퍼를 읽어들이어 프리미티브(삼각형, 점, 선 등의 렌더링 가능한 기하도형. 대개 삼각형)으로 조합하는 역할을 한다.

또한, 시맨틱(시스템 생성 값)을 추가하는 역할도 하며, 이 시맨틱은 다음 단계의 셰이더에서 사용된다.

버텍스 버퍼 : 버텍스들을 모아놓은 버퍼

인덱스 버퍼 : 정점을 사용하여 프리미티브를 만들 때, 프리미티브를 구성하는 정점의 구성을 정의해둔 버퍼. 이를 통해 중복을 제거하여 효율을 높일 수 있다.



Vertex Shader(VS, 버텍스 셰이더)

공간 변환 행렬을 이용하여, 정점을 로컬 공간 -> 투영 공간까지 변환하는 단계이다.

Hull Shader(HS, 헐 셰이더)

Tessellator(TS, 테셀레이터)

Domain Shader(DS, 도메인 셰이더)

테셀레이션을 수행하기 위한 3가지 단계. 모델을 더 세분화한 삼각형인 '패치'로 나누어 더 세밀하게 표현할 수 있게 만든다.

테셀레이션과 변위 매핑을 하기 위한 단계이다.

Geometry Shader(GS, 기하 셰이더)

하나의 프리미티브를 받아서, 새로운 정점을 생성하거나 삭제하거나, 정점을 수정할 수 있다.

그래서 정점 하나만 파이프라인으로 보낸 다음, 거기서 사각형의 Plane(평면)을 만들거나 삼각형을 만들거나 하는 작업도 수행할 수 있다.

Rasterizer(RS, 래스터라이저)

투영 공간까지 변환된 프리미티브를 래스터 그래픽으로 변환시킨다. 래스터 그래픽이란 그림판에 그린 그림을 확대한 것이라고 생각하면 연상이 편한데, '픽셀'을 만들어 주는 것이다.



그림판에 그린 그림을 확대한 모습. 픽셀이 보인다.

즉 도형에 색을 입힐 수 있게 '픽셀'을 채워 주는 것이다.

Pixel Shader(PS, 픽셀 셰이더)

각각의 픽셀마다 수행되며, 픽셀에 하나의 색상을 적용시킨다.

즉 색을 입히는 작업이다.

Output Merger(OM, 출력 병합기)

깊이 버퍼(Depth Buffer)를 통해 깊이 판정을 하여 보이지 않는 일부 픽셀을 폐기한다. 작업 결과르 후면버퍼(Back Buffer)에 쓰고, 알파블렌딩을 수행한다.

1.3.2 조명

조명은 크게 두 가지로 나눌 수 있다. 모든 빛의 반사를 계산하는 전역 조명(GI, Global Illumination)이 있고, 조명 알고리즘을 사용하는 방법이 있다.

GI(전역 조명이란 말보다 글로벌 일루미네이션, GI라는 말을 훨씬 많이 씀)는 두 가지로 나뉜다. 실시간으로 빛의 반사를 계산하는 레이트레이싱, 미리 조명을 모두 계산한 '라이트맵'을 만드는 라이트맵 방식이 있다. 레이트레이싱은 실시간으로 수행해야 하므로 최근에야 실용화가 될랑말랑 하는 기술이다. 라이트맵은 많이 쓰이나, 미리 조명을 계산하여 저장해 놓은 값을 쓰는 방식이므로 맵의 오브젝트의 위치를 바꾼다거나, 빛의 방향을 바꾼다거나 할 수 없다.

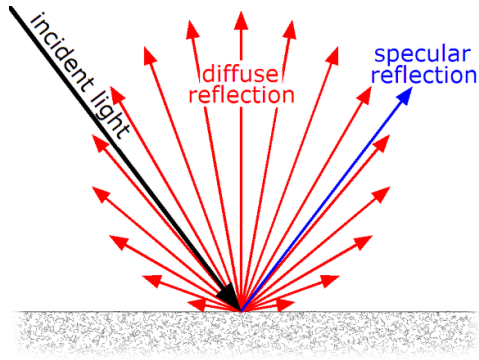
조명 알고리즘은 앰비언트 라이트(주변광, 환경광), 디퓨즈 라이트, 스펙큘러 라이트 셋으로 나뉜다. 중요하진 않지만 이는 품-셰이딩 방법이라고 하며, 정확히는 빛을 만드는 것이 아니라, 빛이 만드는 색을 계산하는 방법이다.

Ambient Light(환경광)

글로벌 일루미네이션에서 환경광이란 빛이 직접 닿지는 않지만, 빛의 반사를 통해 빛이 어느정도 간접적으로 닿는 것을 의미하지만, 조명 알고리즘을 사용할 때는 이걸 구현할 수 없기 때문에 그냥 빛이 어느정도 '있을 것이다' 라고 가정하고 구현한 것을 의미하며, 그렇기 때문에 상수값을 더하여 구현한다.

Diffuse Light(난반사광)

빛이 직접 닿는 곳의 휘도를 구하는 작업이다.

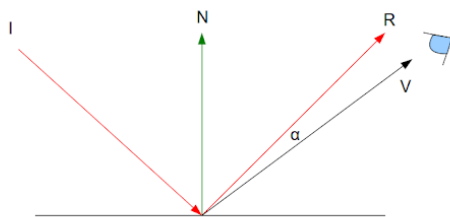


빛은 수직으로 내리쬘 때 가장 밝고, 사선으로 비스듬히 비출 때 가장 어둡기 때문에, 0도에 가까워질수록 작아지고 90도에 가까워 질수록 커지는 **cosine**을 사용하여 값을 구한다.

$$I = (I_i)(kd) \cos \theta = (I_i)(kd)(N \cdot L)$$

Specular Light

반사된 빛이 눈에 직격에 가깝게 닿을 때 그 부분이 하이라이트 되어 밝게 빛나는 것을 말한다.



그렇기 때문에 반사벡터를 구하는 작업이 필요하며, 그래서 반사벡터 구하는 방법을 많이들 물어본다.

1.3.3 뱀프 매핑

<https://www.slideshare.net/SukwooLee4/bump-mapping-164480242>

참조

2. 실제 면접 질문 사례 모음

N:M에서 앞은 면접관, 뒤는 면접자임

2.1 1차면접 모음

2.1.1 N모사 수시채용(RPG)

- 1차 실무, 인성면접(4:1)

- 학원에서 배우면서 가장 어려웠던 과목은 무엇이며, 이유는 무엇인가?
- 학교에서 가장 잘 배운 과목은 무엇인가?
- 자기 포트폴리오 소스코드를 보고 포폴에 대해 설명해보시오

2.1.2 M모사 수시채용(PC, IP기반 신작 RPG)

- 1차 실무, 인사면접(2:1) -> 2차 기술 임원면접(1:1)

- 락에 대하여 아는 대로 설명해 보시오(락, 뮤텍스, 데드락, 세마포어 등..)
- STL의 컨테이너(벡터, 리스트, 맵, 해시, 스택, 큐 등)에 대해 아는대로 설명해 보시오
- 인생게임은 무엇인가요?
- 5년, 10년 후 자신의 모습을 설명해 보세요
- 취미가 무엇인가요?
- 현재 사회적으로 이슈가 되고 있는 일베, 메갈 이런 것들에 대해 어떻게 생각하나요?

2.1.3 D모사 수시채용(퍼즐, 모바일)

- 1차 실무 팀면접(3:1)

- 스마트 포인터에 대해 잘 알고 있는가?
- 데드락을 경험해보았는가? 경험해보았다면 어떻게 해결하였는가?

- 유니티로 카메라를 구현하였던데, 선형보간과 구면보간의 원리를 알고 있는가?
- 코딩을 하면서 가장 어려웠던 사례를 설명해 보세요
- C#과 언리얼의 가비지 컬렉션의 원리를 아는가? C++의 레퍼런스 카운트와의 차이는 무엇인가?
- 협업시에 트러블이 일어난 적이 있는가? 어떻게 해결하였는가?
- 피킹을 구현하였던데, 피킹의 원리에 대해 설명할 수 있는가?
- D모사 면접의 경우, 거의 포트폴리오와 팀플경험에 기반한 질문이었으므로 같은게 나오진 않을 것이라 생각함.

2.1.4 A모사 수시채용(액션 RPG)

- 1차 기술팀면접(2:1) -> 2차 인사면접(2:1)
- C11, C14에서 새로 추가된 STL의 기능들을 아는가?
- 협업시에 트러블이 일어난 적이 있는가? 어떻게 해결하였는가?
- 클라이언트 개발자를 지망하는 이유
- 게임개발자로서의 포부, 목표

2.1.5 N모사 수시채용(스포츠 게임)

- 기술, 임원면접(3:1, 면접 한번으로 채용)
- 팩토리얼을 다양한 방법으로 구현해 보시오(3가지)
- 상대와 내 캐릭터가 있을 때, 상대가 내 왼쪽에 있는지, 오른쪽에 있는지 , 반경 60도 내에 존재하는지 구할 방법을 말해보세요
- 포물선 공식을 아시나요? 가속도에 대해 아시나요?
- 충돌을 구현해보았나요? 어떻게 구현했나요?
- 자신이 다른 지원자 대비 뛰어난 점이 무엇이라 생각하나요?

2.1.6 N모사 수시채용(모바일 IP RPG)

- 1차 실무팀면접(3:3)

- 내가 여태까지 개발하면서 와 이걸 내가 개발하다니 말도 안된다. 할 정도로 감탄스러웠던 경험 있는가?
- 최적화를 수행해 본 경험이 있으면 말해보세요.
- 프로파일링 툴을 써 보았나요? 프로파일링, 최적화를 통해 성능 문제를 해결해 본 경험을 말해보세요
- 자신의 성격이나 인성적 장점을 어필해 보세요
- 벌컨과 메탈에 대해 아시나요?
- 모바일게임을 플레이하면서, 혹은 자사게임을 플레이하면서 기술적으로 인상깊었던 부분이 있나요?
- 스택, 데이터, 힙 메모리에 대해 설명해 보세요
- 왜 게임개발자를 선택하였나요?
- 게임 좋아하세요?
- for문에서 i++(후위연산자)와, ++i(전위연산자)의 차이점이 있나요? 속도차이가 날까요?

2.1.7 N모사 공개채용(액션 RPG)

- 1차 실무팀면접(3:1) -> 2차 부서 임원면접(1:1) -> 3차 인사면접(1:1)
- 동기화 구조체에 대해 설명해 보시오
- 추상클래스는 무엇이며 왜 쓰는가?
- 사용해본 STL 컨테이너들에 대해 설명해 보시오(벡터, 리스트 등...)
- Nagle 알고리즘에 대해 설명해 보시오
- 서버 프로그래밍, 소켓 프로그래밍을 해 보았는가? 무엇을 구현하여 보았는가?
- 우리가 왜 당신을 뽑아야 하는가?
- 가상 메모리에 대해 설명해 보시오
- 왜 게임 개발을 하려고 하는가?
- 32비트 환경과 64비트 환경은 무슨 차이가 있는가?
- TCP와 UDP의 차이점을 설명해 보시오
- 상속구조에서 virtual 소멸자를 써야 하는 이유는 무엇인가?
- 최적화/프로파일링 툴 사용 경험이 있는가? 어떻게 문제를 해결하였는가?

- 기억나는 버그가 있는가?
- C++의 4가지 캐스팅에 대해 설명해 보시오

2.1.8 V모사 수시채용(모바일)

- 1차 실무 팀면접(4:1)
- 당신은 리더인가 팔로워인가?
- 우리 게임에서 기술적으로 인상깊었던 부분은 무엇인가?

2.1.9 A모사 수시채용(액션 RPG)

- 1차 실무팀면접(3:1) -> 2차 기술 임원면접(1:1)
- C++의 4가지 캐스팅에 대해 설명해 보시오
- 알파 블렌딩, 알파 테스트, 알파 소팅에 대해 설명해 보시오
- 라이팅에 대해 설명해 보시오
- Nagle 알고리즘에 대해 설명해 보시오
- TCP와 UDP의 차이에 대해 설명해 보시오
- 언리얼에서 액터와 폰의 차이에 대해 설명해 보시오
- 메모리 단편화에 대해 설명해 보시오
- 당신의 성격적 장점에 대한 근거나 사례가 있는가?
- 스터디를 했던데, 배운 게 정확히 무엇인가? 설명할 수 있는가?
- const의 4가지 예시를 주고 설명하게 함
- 인생에서 열심히 한 게임은 무엇이며, 버그를 찾아본 적이 있는가?
- 반사벡터 구하기
- 언리얼의 스마트 포인터에 대해 아는가?
- IK(Inverse Kinematics)에 대해 들어본 적이 있는가?

2.1.10 P모사 수시채용(PC RPG)

- 1차 실무팀면접(3:2)

- 팀플을 하면서, 프로그래머가 여럿이면 생길 수 있는 다양한 문제들이 있는데 그런 문제들이 있었는지 (순수 기술적인 부분에서)
- 가장 어렵게 디버깅한 경험에 대해 설명하십시오
- 왜 우리팀을 지원하였고, 왜 우리팀에 어울린다고 생각하는가?
- 우리 회사의 인재상이 무엇인가? 왜 당신이 우리 인재상에 맞는 인재라고 생각하는가?
- 다이내믹 캐스트와 스테틱 캐스트의 차이는 무엇인가? 주의점은 무엇인가?
- 애니메이션을 구현하였던데, 애니메이션에 대해 설명해 보시오
- 왜 클라이언트 포지션을 선택하였는가? 왜 이 포지션을 잘 할 것이라 생각하는가?
- 프로젝트에서 어떤 파트를 맡았는가? 그 파트를 어떻게 구현을 하였는가?(순수 기술적인 부분만)
- 자주 사용하는 자료구조가 있는가? 하나를 설명해보세요(트리, 맵, 그래프, 해시 등..)
- 메모리 단편화의 두 가지 경우를 설명하고, 해결 방법을 제시하십시오
- 상사가 자기가 보기에 무의미한 일(6개월 내내 로그만 찍는 것 같은)을 지시한다면, 어떻게 할 것인가?
- 본인의 장점과 단점은 무엇인가?
- 필기 테스트 문제 중에 지금 생각해보니 잘못 썼다고 생각하는 문제가 있는가? 어떻게 고칠 것인가?

2.1.11 K모사 공개채용(PC RPG)

- 1차 실무팀면접(2:1)

- 프로세스와 스레드에 대해 설명하십시오
- 컨텍스트 스위칭에 대해 설명하십시오
- 메모리 단편화에 대해 설명하십시오
- 게임 개발자를 목표로 하게 된 특별한 계기가 있는가?
- 자신의 장/단점에 대해 설명해 보시오

2.2 2차면접 모음

2.2.1 A모사 수시채용(액션 RPG)

- 2:1 면접

- 가장 행복했던 순간 / 불행했던 순간에 대해 말해 보시오
- 열정을 다했던 경험이 있는가?
- 자기가 다른 지원자보다 뛰어난 점이 무엇인가?
- 자기 PR 해보세요
- 우리 게임 해 봤는가? 왜 안해봤는가?
- 자신을 한마디로 표현하면?
- 스트레스 해소는 어떻게 하는가?
- 인사팀에 물어볼 것이 있는가?
- 자신은 리더인가 팔로워인가?
- 자기가 일하고 싶은 좋은 회사란 어떤 회사라고 생각하는가?
- 팀원과 대립이 생기면 어떻게 해소하려 하는가?
- 왜 게임개발자를 지망하는가?
- 지금 즐기는 게임 / 모바일 게임 이 있는가?

2.2.2 N모사 공개채용(액션 RPG)

- 1:1 면접

- 프로젝트에서 무슨 파트를 담당하였는가?
- 인턴(실습)경험이 있던데, 그 회사 대표가 너를 어떻게 평가할 것 같은가?
- 왜 우리 회사이며, 만약 우리 회사에서 떨어진다면 어떻게 할 생각인가?
- 우리가 왜 너를 뽑아야 하는가?
- 하고 싶은 말이 있는가?
- 동료들이 너를 어떻게 생각할까? 뭐라고 평가할까?

- 게임 개발자로서 ~한 작업을 주로 하였고, 하고 싶다고 썼는데 원하는 작업을 못 하게 될 수도 있다. 그럴 경우 어떻게 할 것인가?

2.2.3 M모사 수시채용(PC, IP기반 신작 RPG)

- 1:1 기술 임원 면접

- 도전을 해 본 경험이 있는가?

- 자신의 장점과 단점을 말하고, 단점을 어떻게 극복하고 있는지를 객관적 근거를 들어 말하시오. (객관적 근거란 만약 코딩이 느리다가 단점이면 시간 측정을 해본다거나 주변 동료들에게 물어본다거나 원인 분석을 해본다거나 같은 다양한 근거를 말한다)

- 객체지향과 절차지향의 차이점을 설명하고, 왜 객체지향을 쓰는지를 설명하시오.

- 다형성에 대해 설명하시오.

- 소켓 프로그래밍을 이용한 서버를 구현하여 게임을 만들었던데, 패킷이 어떤 식으로 동작하는가? 만약 캐릭터가 대기 상태라면 패킷이 전송되는가?

3. 주관적 팁 모음(주관성, 비정확성 주의)

3.1 1차면접의 패턴

경험상 1차면접은 두 가지 패턴으로 나뉜다고 생각합니다.

팀 프로젝트와 개발의 '경험'을 주로 물어보는 면접과, 이론적인 '지식'과 테크닉을 물어보는 면접입니다. 이것은 면접관 취향에 따라 케바케가 워낙 심한 부분이기 때문에 두 부분을 다 준비하여야 합니다.

다만, 공개채용의 경우 전공자를 선호하는 경향이 있어서 컴퓨터 아키텍처, 시스템 프로그래밍, OS, 네트워크 같은 전공지식적인 질문이 들어오는 경향이 있습니다만 대개 어느정도 고만고만한 질문이 들어오기 때문에 출제 패턴을 파악해두면 대처하기 쉽다는 생각이 듭니다.

3.1.1 경험을 묻는 질문의 예시

경험을 묻는 질문의 예시는 다음과 같습니다.

협업 프로젝트 경험

프로젝트 수행 시에 팀원들과 대립하거나 의견이 엇갈렸을 때 해결한 경험

최적화를 수행한 경험

성능 문제가 생겼을 때 프로파일링 툴이나 기타 방법을 사용해서 해결한 경험

역경을 극복한 경험

뛰어난 발상이나 창의력으로 문제를 해결한 경험

가장 어렵게 디버깅한 경험

개발하면서 가장 힘들었던 경험

기술적인 문제에 마주쳤던 경험

어떠한 일에 열정을 다 해본 경험

어떠한 일에 도전해서 성취를 느껴본 경험

3.1.2 이러한 질문에 대처하려면

이러한 경험은 프로젝트 시에 미리 신경써서 해당 문제를 기억해 놓아야 하고, 없으면 만들어서라도 채워 두는게 좋습니다. 특히 프로파일링, 최적화 경험 같은건 실제로 한번 해 보아야 하기 때문에 프로젝트 중이 아니면 경험하기가 어렵습니다.

또한 역경, 도전, 커뮤니케이션 경험 같은 항목들은 자신의 인생을 되돌아보며 그럴듯해보이는 이야기에 살을 붙여서 약간의 소설을 쓰는게 좋습니다. 인생이 무난한 저 같은 사람들은 대개 그렇게 굴곡진 인생을 살아오지 않았기 때문입니다. 그렇다고 너무 픽션 비율이 높으면 추가질문이 들어올 수 있으니 픽션/논픽션 밸런스에 주의합니다.

3.1.3 지식과 테크닉을 묻는 질문의 예시

상속으로 구현한 부모 클래스의 소멸자에 반드시 **virtual**이 들어가야 하는 이유는?

Nagle 알고리즘에 대해 설명해 보시오

반사벡터를 구하는 방법을 설명해보시오

TCP와 **UDP**의 차이점은?

Starvation(기아 상태)에 대해 설명하시오

렌더링 파이프라인에 대해 설명해 보시오

volatile이란 무엇인가?

프로세스와 스레드에 대해 설명하시오

포워드 렌더링과 디퍼드 렌더링에 대해 아는가?

3.1.4 이러한 질문에 대처하려면

이러한 질문들은 공채에서 자주 나오는 질문이지만, **C++** 테크닉의 경우 수시채용에서도 꽤나 들어오는 질문입니다. 면접관 취향에 따라 케바케가 극심하기 때문에 안 묻는 분들은 한마디도 안 묻고, 묻는 분들은 이것만 묻는 분들도 있기 때문에 준비가 필요한 부분입니다.

대개 **C++** 테크닉, 컴퓨터 공학, 컴퓨터 그래픽스에 대한 질문이므로 이 세가지 부분을 미리 공부해두는 것이 좋습니다.

Effective C++(면접에서 나오는 **C++** 테크닉은 이걸로 대충 다 커버가 됩니다), 운영체제, 컴퓨터 구조(정말 가야끔 나옵니다), 네트워크(대개 **TCP/UDP** 근처 부분만), **Tbasis 3D**와 관련 **DirectX** 책의 정독을 권합니다.

3.3 그 외 팁

이 부분은 면접 책에 나온 내용이거나, 직접 겪으면서 아 이걸 하면 안되겠구나 싶은 것들을 모아보았습니다.

회사는 배우는 사람이나 발전할 예정인 사람을 별로 안 좋아한다고 합니다.

~신입이라 아직 부족하지만, ~아직 부족한 것이 많고 모르는 것이 많지만 열심히 배우겠습니다. 아직은
능력이 부족하지만... 같은 멘트는 치지 맙시다.

기업의 비전과 인재상을 숙지하고 갑시다. 가끔 물어봅니다. 내가 왜 인재상에 어울리는지도 생각해
봅시다.

진짜 아무리 생각해도 모르겠다 싶은 질문에는 솔직하게 모르겠습니다 하고 다음 질문으로 넘어갑시다.

질질 끌면 안좋아합니다.

단점, 약점에 대한 질문은 그 약점을 어떻게 극복하려 노력하고 있는지를 묻는 질문이라고 합니다. 뭔가
약점에 대한 질문을 받았을 때(성적이 왜 이렇게 낮나요? 코테를 왜 이렇게 못풀었나요?) 당황하지 말고
인정할 건 인정하고 극복을 위한 노력에 대한 얘기로 풀어나갑시다.

무의식적으로 면접관 눈을 피하지 말고 똑바로 쳐다보고 말합시다.

대답할 때 어~ 씬~ 하는 말버릇이 있으면 교정합시다.