**DFS**

1. 깊이 우선 탐색이란 시작 노드에서 다음 분기로 넘어가기 전에 해당 분기를 완벽하게 탐색하는 방법이다.
2. 자기 자신을 호출하는 순환 알고리즘의 형태를 가지고 있다.
3. 어떤 노드를 방문했는지 여부를 검사하지 않는다면 무한루프에 빠질 수 있다.
4. 모든 노드를 방문 하고자 하는 경우에 사용한다.
5. 스택을 사용한다.

**BFS**

1. 너비 우선 탐색이란 시작 노드에서 인접한 노드를 먼저 탐색하는 방법이다.
2. 시작 정점으로부터 가까운 정점을 먼저 방문하고 멀리 떨어져 있는 정점을 나중에 방문하는 순회방법이다.
3. 보통 두 노드 사이의 최단 경로 혹은 임의의 경로를 찾고 싶을 때 사용한다.
4. 재귀적으로 동작하지 않는다
5. 어떤 노드를 방문했었는지 여부를 반드시 검사해야 하고 하지 않을 시 무한 루프에 빠질 수 있다.
6. BFS는 방문한 노드를 차례로 저장한 후 꺼낼 수 있는 큐를 사용한다.

**BFS & DFS**

1. 깊이 우선 탐색이 너비 우선 탐색보다 구현이 간단하다.
2. 단순 검색 속도 자체는 너비 우선 탐색에 비해서 느리다.

**다형성**

\*\* 다형성이란 여러 개의 서로 다른 객체가 동일한 기능을 서로 다른 방법으로 처리할 수 있는 기능을 의미한다.

**Virtual**

C++에서 상속 관계가 있으며 메모리 해제를 해야 하는 경우 반드시 부모 클래스의 소멸자를 가상 함수 선언해야 한다.

만약 그렇지 않으면 자식 클래스의 소멸자는 호출되지 않고 부모클래스의 소멸자만 호출되어 자식 클래스의 객체가 해제되지 않는 현상이 발생한다.

**순수 가상 함수**

C++의 가상 함수는 기본적으로 반드시 재정의할 필요는 없지만 순수 가상함수는 자식 클래스에서 반드시 재정의를 해야 한다.

그래서 일반적으로 순수 가상 함수는 부모 클래스에서 함수 동작을 정의하지 않는 대신 자식 클래스에서 반드시 정의해야 사용 가능하다.

추상 클래스란 하나 이상의 순수 가상 함수를 포함하는 클래스를 의미하며 다형성을 효과적으로 프로그램 상에서 구현할 수 있다.

**constexpr**

constexpr 키워드는 객체나 함수 앞에 붙는 키워드로 해당 객체나 함수의 리턴값을 컴파일 타임에 알 수 있다 라는 의미를 가진다.

해당 키워드를 잘못 사용할 시 컴파일 타임에 오류가 발생하기 때문에 런타임에 오류를 확인 하는 것 보다 좋다.

**추상클래스와 인터페이스**

추상 클래스

클래스 구현부 내부에 추상 메서드가 하나 이상 포함되거나 abstract로 정의된 경우를 말한다.

특징

1. New 연산자를 사용하여 객체를 생성할 수 없다.
2. 추상 클래스(부모)와 일반 클래스(자식)는 상속의 관계에 놓여있다.
3. 새로운 일반 클래스를 위한 부모 클래스의 용도로만 사용된다.
4. 일반 클래스들의 필드와 메소드를 통일하여 일반 클래스 작성 시 시간을 절약할 수 있다.
5. 추상 클래스는 단일 상속만 가능하며 일반 변수를 가질 수 있다.
6. 추상 클래스는 동일한 부모를 가지는 클래스를 묶는 개념으로 상속을 받아서 기능을 확장시키는 것이 목적이다.

인터페이스

모든 메서드가 추상 메서드 인 경우를 말한다.

특징

1. 인터페이스는 추상 클래스보다 한 단계 더 추상화 된 클래스 이며 모든 메서드 들이 추상 메서드로 간주되기 때문에 abstract를 적지 않는다.
2. 인터페이스는 static final필드만 가질 수 있다. 필드를 선언할 때는 public static final이 생략 되어있다고 생각하면 된다.

* Public static final을 사용하는 이유

구현 객체의 같은 동작을 보장

인터페이스의 변수는 스스로 초기화 될 권한이 없음

아무 인스턴스도 존재하지 않는 시점이기 때문

**추상 클래스와 인터페이스의 공통점과 차이점**

공통점

1. 선언만 있고 구현 내용은 없는 클래스이다.
2. 인스턴스화(객체를 생성)할 수 없다.
3. 추상 클래스를 extends로 상속받은 자식들과 인터페이스를 implements하고 구현한 자식들만 객체를 생성할 수 있다.

– 결국 자식 클래스가 무언가 반드시 구현하도록 위임할 때 사용한다.

차이점

1. 추상 클래스는 단일 상속이고 인터페이스는 다중상속이 가능하다.
2. 추상 클래스의 목적은 상속을 받아서 기능을 확장시키는 것(부모의 유전자를 물려받는다.)
3. 인터페이스의 목적은 구현하는 모든 클래스에 대한 특정한 메서드가 반드시 존재하도록 강제하는 역할(부모로부터 유전자를 물려받는 것이 아니라 사교적으로 필요에 따라 결합하는 관계)즉, 구현 객체가 같은 동작을 한다는 것을 보장하기 위함.

**스마트 포인터**

스택 메모리는 scope를 벗어나면 자동으로 사라지는 특성을 힙 메모리에 적용한 것이 RAII라는 개념이다.

프로그래머의 실수로 delete를 안 하면 Memory leak이 발생하게 되는데 이를 방지하기 위해 만만들 진 것이 스마트 포인터 이다.

1. Unique\_Ptr
2. 원시 포인터(소유 중인 포인터)의 소유자는 한 명이다.
3. 원시 포인터에 대한 소유권을 이전할 순 있지만 복사나 대입과 같은 공유를 불허한다.
4. 유니크 포인터 객체가 소멸될 때 원시 포인터도 소멸된다.

유니크 포인터는 멤버 함수인 reset을 통해 현재 소유 중인 원시 포인터를 삭제 또는 다른 원시 포인터로 재설정 할 수 있다.

Release를 사용하여 소유하던 원시 포인터를 해제하지 않고 외부로 내보낼 수 있다.

* 유니크 포인터의 단점

원시 포인터에 대한 소유권을 다른 유니크 포인터와 공유할 수 없다. 즉 복사생성과 복사 대입 연산이 불가능하며 공유를 허용하지 않았다.

그래서 함수의 인자로 포인터를 넘겨야 할 경우 get을 통해 원시 포인터를 반환하는 방식으로 넘겨야 했다.

그러나 이는 외부에서 마음대로 지워버릴 수 있기 때문에 위험했으며, 원시 포인터를 반환하여 다른 곳에서 사용한다는 것 자체가 스마트 포인터를 사용하는 이점을 감소시키는 것이기도 했다.

또는 공유가 아닌 이동을 통해 자신의 소유권을 포기하고 다른 유니크 포인터에 넘기는 것도 하나의 방법이었다. 하지만 포인터를 함수에 전달하는 경우는 자주 발생하기에 이에 적합한 스마트 포인터가 존재하는데 그것이 Shared\_ptr이다.

**스마트 포인터**

1. **Shared\_ptr**
2. 원시 포인터에 대한 소유권을 다른 공유 포인터와 공유할 수 있다.
3. 참조 카운팅을 기반으로 동작하는 스마트 포인터이다.
4. 복사 생성과 복사 대입 연산 모두 허용한다.

* 멀티스레드 환경에서의 문제점

우선 이 문제는 참조카운트 방식의 문제이지만 공유 포인터에서는 해결되었다.

예를 들어 참조카운트가 1인 메모리 M을 스레드 A,B가 참조하려고 할 때

두 스레드가 M이 1인 상태로 가져가서 사용한다면 참조 카운트가 2가 되기 때문에 의도했던 3과는 다른 결과가 나오게 된다.

이를 경쟁 상태라고하는데 LOCK이나 MUTEX와 같은 함수를 통해 두개 이상의 스레드가 동시에 접근하지 못하도록 막아야한다.

* 참조 카운팅 방식의 문제점

순환 참조를 하는 경우가 생기는데 순환 참조란 참조 횟수를 가지는 두 개체가 서로를 참조하고 있어 참조 카운트가 0이 되지 못해 스스로 메모리 해제가 불가능한 상황을 말한다.

이를 해결하기 위해 weak\_ptr이 존재한다.

**스마트 포인터**

Weak\_ptr

Weak는 유니크와 공유 포인터와는 다르게 단독으로 혼자 사용할 수 없다.

weak포인터는 원시 포인터 대신 공유포인터를 건네받아 공유 포인터가 소유한 원시포인터를 참조하도록 한다.

즉 직접적으로 원시 포인터를 인자로 받을 수 없으며 \*, -> 를 통해 포인터 처럼 사용할 수도 없다.

Weak 포인터가 공유 포인터의 원시 포인터를 참조할 때는 약한 참조를 통해 참조하므로 약한 참조 횟수가 증가 한다. (공유 포인터 간에는 강한 참조)

* 강한 참조와 약한 참조의 차이

각 참조의 횟수가 메모리 해제에 영향을 미치느냐 안 미치느냐 이다.

공유 포인터에선 참조 횟수가 0이 되면 알아서 소멸자를 호출하여 메모리를 해제한다.

이 때의 참조 횟수는 바로 강한 참조이고 학한 참조 횟수는 메모리 해제와 관련이 없다. 약한 참조가 100,200이여도 강한 참조 횟수가 0이 되면 원시 포인터의 메모리는 해제된다.

* Weak ptr는 자신이 참조하는 원시 포인터의 강한 참조 횟수가 0이되면 expired 라는 상태가 되는데 이는 원시 포인터의 강한 참조 횟수가 0이 되어 메모리가 해제되었으므로 weak ptr은 더 이상 유효하지 않다 라는 의미이다.
* Expired 멤버 함수

expired라는 함수가 존재 하는데 이는 참조하던 원시 포인터가 소멸되었다면 true 아니라면 false를 반환한다.

참고로 weak ptr은 reset이 없다 아마 인자로 원시 포인터를 받지 않아서 인 것 같다.

**클래스 & 구조체**

* 공통점

1. 변수, 메소드를 하나로 묶는 방법
2. 변수,함수,생성,소멸자를 내부에서 선언할 수 있고 상속구조를 사용할 수 있다.
3. 접근제어 지시자가 존재하며, 구조체, 클래스 모두 접근제어 지시자를 사용할 수 있다.

* 차이점

1. 구조체는 접근 제어 지시자를 따로 명시하지 않는다면 모든 변수와 함수는 public으로 선언되고 클래스는 private로 선언 된다.
2. 일반적으로 구조체는 public으로 설정하여 외부에서 멤버변수를 쉽게 수정할 수 있게 하지만 클래스는 멤버 변수를 private로 멤버함수는 public으로 선언하여 멤버함수를 통해서만 멤버변수를 수정할 수 있게 설계한다.

* 접근제어 지시자

Public : 어디서든 접근 허용

Protected : 상속 관계에 놓여있을 때, 유도 클래스에서의 접근 허용

Private : 클래스 내에서만 접근 허용

**상속과 포함**

* 상속의 장점

1. 동일한 코드가 여러 클래스에 중복되는 일을 해결해줌
2. 계층적 분류 및 관리가 용이함
3. 기존에 만들어 놓은 클래스를 상속받고 기능을 확장하여 새로운 클래스를 쉽게 만들 수 있다.

* 상속과 포함의 차이점

상속 관계에서 둘은 밀접하게 결합되므로 부모 클래스가 변경되면 코드가 손상될 위험이 있지만 계층구조에서 상위 클래스의 기능을 하위 클래스가 물려받아 사용할 수 있는 장점이 있다.

반면 포함은 느슨하게 결합되어 클래스의 내용이 변경되더라도 구성 요소를 쉽게 변경할 수 있다. 이런 점에서 유연성을 제공하지만

**객체지향 언어 OOP(Object Oriented Programming)**

객체지향 프로그래밍 이란?

프로그래밍에서 필요한 데이터를 추상화시켜 상태와 행위를 가진 객체를 만들고 그 객체들 간의 유기적인 상호작용을 통해 로직을 구성하는 프로그래밍 방법

* 장점

1. 상속을 통해 확장해서 사용할 수 있으므로 코드 재사용에 용이하다.
2. 절차 지향 프로그래밍에서는 코드를 수정해야할 때 일일이 찾아 수정해야하는 반면 객체 지향 프로그래밍에서는 수정해야 할 부분이 클래스 내부에 멤버 변수 혹은 매서드로 있기 때문에 해당 부분만 수정하면 됨
3. 클래스 단위로 모듈화 시켜 개발할 수 있으므로 대형 프로젝트처럼 여러 명, 여러회사 에서 개발이 필요할 시 업무 분담하기가 쉽다는 장점이 있다.

* 단점

1. 처리속도가 상대적으로 느리다.
2. 객체가 많으면 용량이 커질 수 있다.
3. 설계시 많은 시간과 노력이 필요하다.

**객체지향 언어 OOP(Object Oriented Programming)**

특징

1. 추상화

* 객체들의 공통적인 속성과 기능을 중심으로 추상화 한다.

1. 캡슐화

* 실제로 구현되는 부분을 외부에 드러나지 않도록 정보를 은닉할 수 있다.
* 객체가 독립적으로 역할을 할 수 있도록 데이터와 기ㅏ능을 하나로 묶어 관리한다.
* 코드가 묶여있어서 오류가 없어 편리하다.
* 데이터를 보이지 않고 외부와 상호작용을 할 때는 메소드를 이용하여 통신을 한다.

보통 라이브러리로 만들어서 업그레이드해 사용할 수 있다.

1. 상속성

* 하나의 클래스가 가진 특성을 다른 클래스가 그대로 물려받는 것
* 이미 작성된 클래스를 받아서 새로운 클래스를 생성하는 것
* 기존 코드를 재활용 해서 사용함으로써 객체지향 방법의 중요한 기능 중 하나에 속한다.

1. 다형성

* 약간 다른 방법으로 동작하는 함수를 동일한 이름으로 호출하는 것
* 동일한 명령의 해석을 연결된 객체에 의존하는 것
* 오버라이딩과 오버로딩
  + 오버라이딩

부모클래스의 메소드와 같은 이름을 사용하며 매개변수도 같되 내부 소스를 재정의하는 것

* + 오버로딩

같은 이름의 함수를 여러 개 정의한 후 매개변수를 다르게 하여 같은 이름을 경우에 따라 호출하여 사용하는 것

1. 동적 바인딩

* 가상 함수를 호출하는 코드를 컴파일 할 때 바인딩을 실행ㅇ시간에 결정하는 것
* 파생 클래스의 객체에 대해, 기본 클래스의 포인터로 가상 함수가 호출될 대 일어난다.
* 함수를 호출하면 동적 바인딩을 통해 파생 클래스에 오버라이딩 된 함수가 실행
* 프로그래밍의 유연성을 높여주며 파생 클래스에서 재정의한 함수의 호출을 보장한다.

**객체지향 언어 OOP(Object Oriented Programming)**

객체지향 개발의 5대 원칙

1. 단일 책임의 원칙 SRP

* 클래스는 한가지 기능만 가지며 한가지 책임을 수행하는데 집중되어야 한다.
* 여러가지의 책임으로 나눌때는 각 책임간에 결합도를 최소로 하여야 한다.
* 하나의 책임이 여러군데 분산되어 있는 상황에 하나의 수정 사항이 여러 군데에 영향을 미치는 경우 다시 전부 하나로 모아주어 설걔를 깔끔하게 유지 시켜주도록 한다.
* SRP 원칙을 따르면 클래스를 여러가지로 분할하여 유연하게 설계할 수 있는 장점을 가지고 있으며 다른 행동, 책임이 격리 되어있어 연쇄적인 문제가 발생할 여지가 줄어든다.
* 책임을 적절히 분배함으로써 코드의 가독성, 유지보수 에 용이하고 다른 원리들의 기초가된다.

1. 개방폐쇄의 원칙 OCP

* 소프트웨어의 구성요소는 확장에는 열려있고 변경에는 닫혀있어야 한다.
* 변경에 대한 비용을 최대한 줄이면서 확장에 대해서는 가능한 극대화 해야 한다는 것으로 다른 추가 사항이 일어나더라도 기존 구성은 변경하지 않으며 확장에 대한 가능성을 열어줘야 한다는 뜻입니다.

1. 리스코브 치환의 원칙

* 서브 타입은 언제나 기반 타입으로 교체할 수 있어야 한다.
* 상속의 기본적인 매커니즘을 표현하는 원칙으로 사용자 관점에서 기능에 영향을 미치지 않고 서브 클래스를 부모 클래스로 대체 할 수 있어야 한다.

1. 인터페이스 분리의 원칙 ISP

* 한 클래스는 자신이 사용하지 않는 인터페이스는 구현하지 말아야 한다.
* 어떤 클래스가 다른 클래스를 상속받았을 때 최소한의 인터페이스만을 사용해야하는데 ISP는 일반적인 한 개의 인터페이스 보다 구체적이 녕러가지의 인터페이스를 구현한다는 원칙이다.

1. 의존성역전의 원칙 DIP

* 고수준 모듈은 저수준 모듈에 의존해서는 안되며 추상화에 의존해야한다.
* 추상화는 세부 사항에 의존 해서는 안된다.

**디자인 패턴**

1. MVC 패턴

**Malloc & New**

* 차이점

1. Malloc은 해당 포인터의 타입을 모르기 때문에 리턴값의 자료형은 void\*이다.

그래서 malloc 사용시에는 int\* i = (int\*)malloc(sizeof(int))로 자료형을 선언해 주어야하고 sizeof의 도움을 받아야 한다.

하지만 new는 type-safe(컴파일러가 타입형을 확인해 주는 것)로 해당 객체에 맞는 포인터로 반환해 준다.

1. New를 이용해 객체를 생성하면 초기 값을 줄 수 있다.(객체를 자동으로 초기화)

New로 객체를 생성하면 생성자가 호출되지만 malloc은 생성자에 호출 기능이 없기 때문에 초기값을 줄 수 없다.

1. New를 통해 객체생성시 에러가 발생하면 예외처리를 하지만 malloc은 예외처리 없이 NULL값을 반환하게 만든다.
2. Malloc은 realloc으로 할당된 메모리 크기를 재조정이 가능하지만 new는 할당된 크기에 대한 메모리 재조정이 불가능하다.

new에서는 새로할당 -> 복사 -> 해제 하는 과정을 거쳐야 한다.

* Delete와 free의 차이점
  + 메모리를 해제한다는 점은 같지만 delete는 소멸자를 호출해 준고 free는 소멸자를 호출해 주지 않는다.
* Malloc의 동작 원리

보통의 경우 미리 할당된 메모리풀을 사용하며, 프로그램 실행 초기에 만들어주고 malloc은 사용자 영역에 생성된 메모리 풀의 메모리를 할당해 준다.

그리고 생성에 실패하면 NULL 포인터를 반환해서 확인하지 않고 포인터를 사용하면안된다.

**메모리 영역**

* 코드 영역

1. 우리가 작성한 소스 코드가 기계어 형태로 저장된다.
2. 컴파일 타임에 결정되고, 중간에 코드가 변경되지 않도록 Read-Only 형태로 저장된다.

* 데이터 영역

1. 전역변수, static 변수가 저장된다.
2. 프로그램 시작과 동시에 할당되고, 프로그램이 종료 되어야 메모리가 해제된다.
3. 실행 도중 병수 값이 변경될 수 있으니 Read-Write로 지정된다.

* 힙 영역

1. 프로그래머에 의해 메모리 공간이 동적으로 할당되고 해제된다.
2. 메모리의 낮은 주소에서 높은 주소으,ㅣ 방향으로 할당된다.
3. 런타임 시에 크기가 결정된다.
   * 힙의 장점
4. 개체의 개수나 크기를 미리 알 수 없는 경우 사용 가능하다.
5. 개체가 너무 커서 스택 할당자에 맞지 않는 경우 사용 가능하다.
   * 힙의 단점
6. 할당 작업으로 인해 속도가 저하 될 수 있다.
7. 이중 해제, 해제 후 블록 사용, 블록 경계를 벗어나 덮어쓰는 등의 문제 발생시 속도가 저하됨
8. 힙 경합으로 인한 속도저하

두 개 이상의 스레드에서 동시에 데이터에 액세스 하려고하면 경합이 발생하여 한 쪽 스레드의 작업이 완료되어야 다른 쪽 스레드의 작업이 지행 될 수 있고 이 문제는 다중 프로세스 시스템에서 일어나는 큰 문제중 하나입니다.

이를 해결핮기 위해 해당 힙을 사용하는 모든 작업을 시리얼라이즈하는것인데 스레드에서는 잠금을 기다리는 동안 컨텍스트 스위칭 할 수 있습니다.

**메모리 영역**

* 스택 영역

1. 함수의 호출과 관계되는 지역 변수와 매개변수가 저장되는 영역이다.
2. 스택 영역은 함수의 호출과 함께 할당되며 함수의 호출이 완료되면 소멸한다.
3. 메모리의 높은 주소에서 낮은 주소의 방향으로 할당된다.
4. 스택이 클수록 힙이 작아지고 힙이 클수록 스택이 작아진다.
5. 스택영역은 컴파일 시에 크기가 결정된다.

* 장점

1. 낭비되는 공간이 없다.
2. 하나의 명령으로 메모리 조작과 어드레스 조작이 가능하다.

* 단점

1. 한계가 있어 한계를 초과하도록 삽입할 수 없다.
2. 유연성이 부족하다.

**메모리 영역**

* 스택 할당 속도 vs 힙 할당 속도

1. 스택이 훨씬 빠르다.

스택은 이미 할당 되어 잇는 공간을 사용하는 것이고 힙은 따로 할당해서 사용하는 공간이기 때문에 스택이 훨씬 빠르다.

1. 하지만 스택은 공간이 매우 적기 때문에 모든 응용 프로그램에서 스택을 사용할 수는 없다.

* 좀 더 자세히 알아보자

스택에서 할당의 의미는 이미 생성되어 있는 스택에 대해 포인터의 위치만 바꿔주는 단순한 연산이다.

반면 힙에서의 할당은 요청된 크기, 현재 메모리의 상황 등 다양한 요소를 고려하기 때문에 스택이 빠르다.

* 스택과 힙의 저장
* 스택의 지역변수는 사용되고 소멸하므로 데이터 용량의 불확실하다. 때문에 밑에서부터 채워 올리고
* 힙은 위에서부터 채워진다.