
C++ 학술회
1주차 상속 / 캐스팅 / 싱글톤
발표자 : 방찬웅

01

간단한 소개

02

상속

03

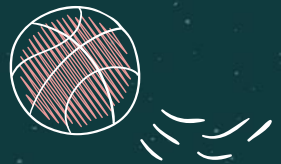
캐스팅

04

싱글톤

상속

02.



상속이란?

일정한 친족적 관계가 있는 사람 사이에 한쪽이 사망하거나
법률상의 원인이 발생 하였을 때
재산적 또는 친족적 권리와 의무를 계승하는 제도

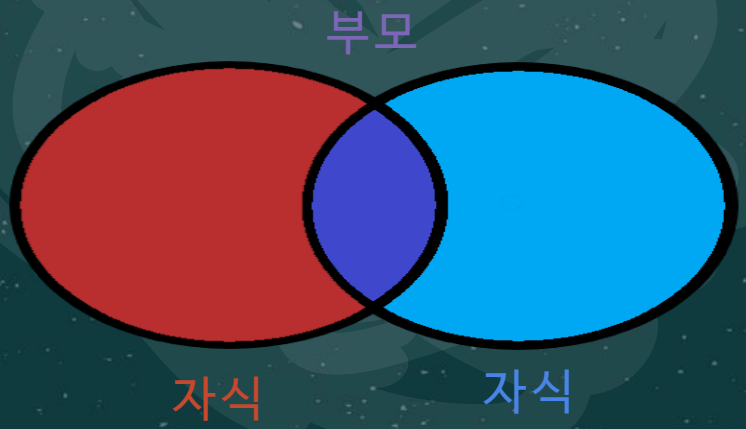
STUDY HARD!

상속이란?

클래스 간의 부모 자식 관계를 정하고,
자식 클래스가 부모 클래스의 모든 속성을
그대로 물려받아 사용할 수 있게 해주는 것.

쉽게 말하자면!

부모 클래스는 자식 클래스들의 공통점을 가진다!



왜 쓰는가?

사용자에게 높은 수준의 코드 재활용성 제공

클래스간의 계층적 관계를 구성해, 다형성의 문법적 토대 마련

다형성

Ploymorphism

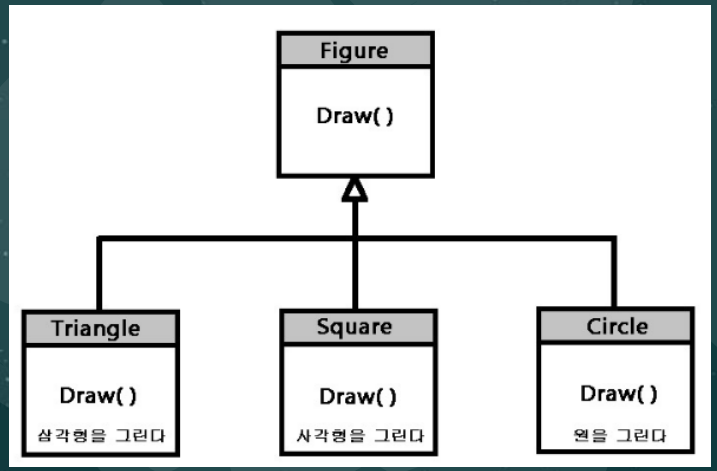
하나의 객체가 여러가지 타입을 가질 수 있는 것!

예) GameObject가 Map으로!

부모가 자식에 빙의한다.

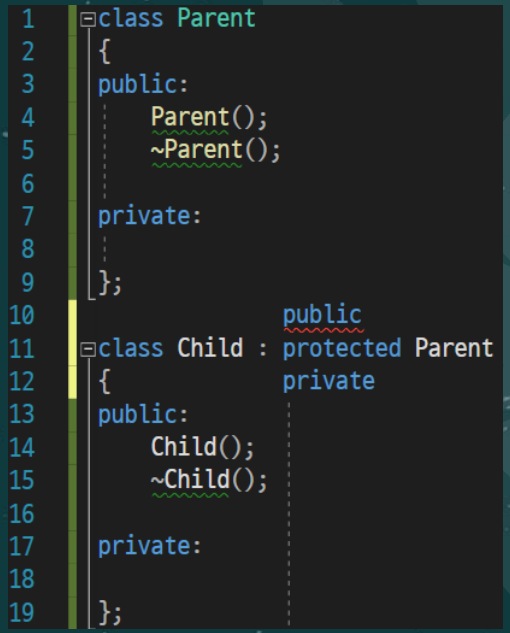
다형성

Ploymorphism



Figure은 삼각형도 될 수 있고
사각형도 될 수 있고
원도 될 수 있다!

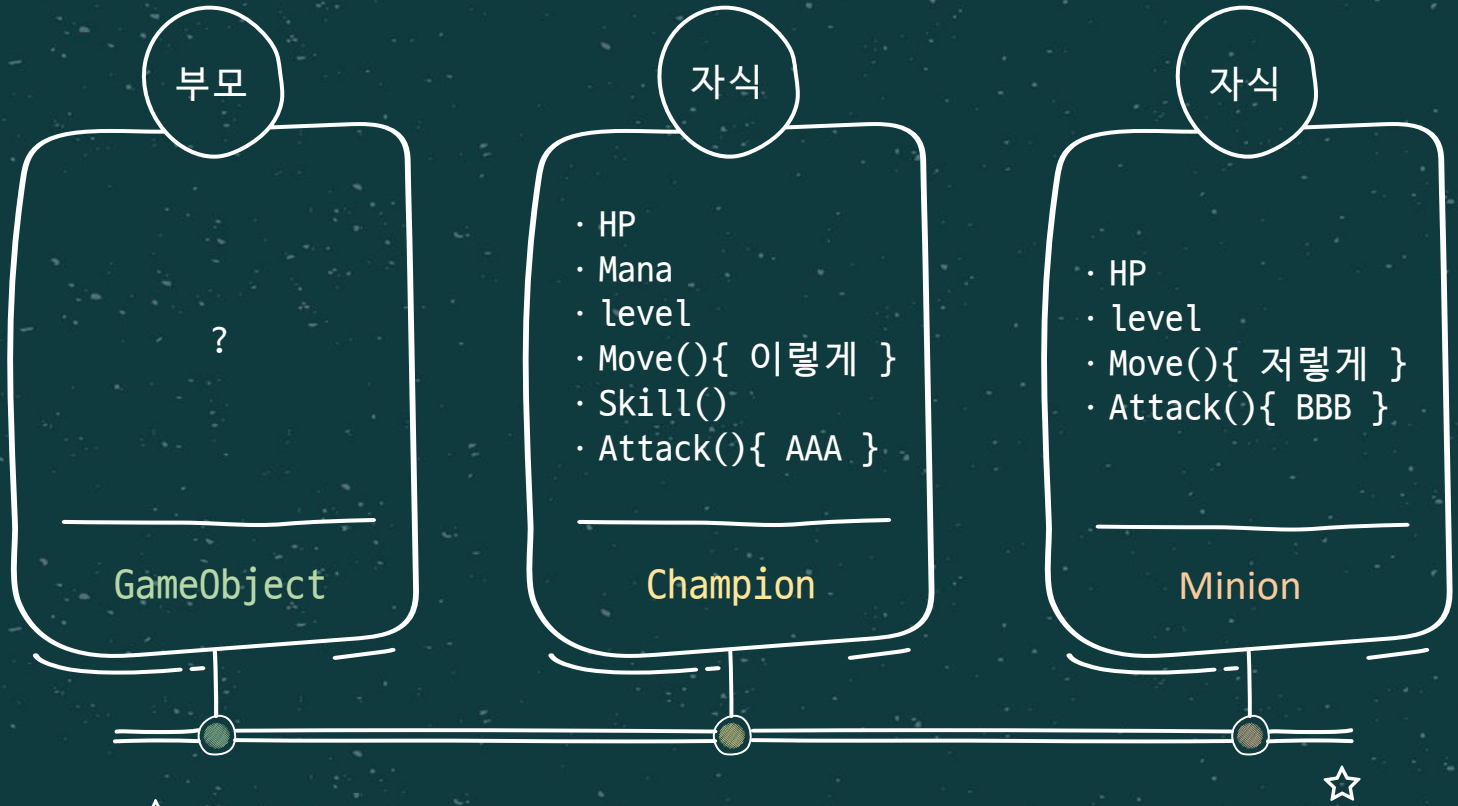
정의 및 접근 지정자



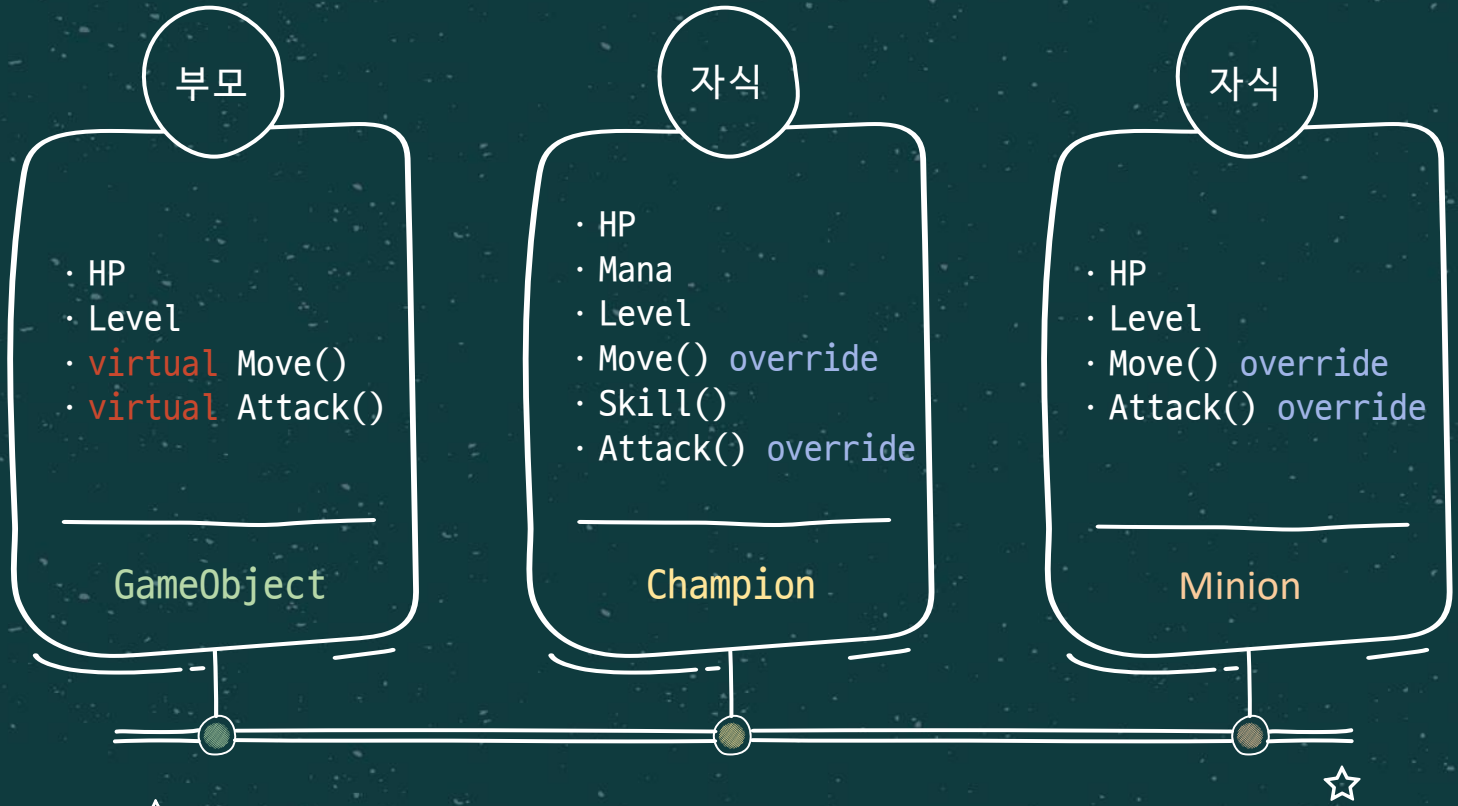
상속 접근 지정자	기반 클래스	파생 클래스로의 상속형태
public	public	public
	private	접근 불가
	protected	protected
private	public	private
	private	접근 불가
	protected	private
protected	public	protected
	private	접근 불가
	protected	protected

접근 지정자 별 접근 형태

만약 LOL 이라면


부모
?
GameObject
자식
· HP
· Mana
· level
· Move(){ 이렇게 }
· Skill()
· Attack(){ AAA }
Champion
자식
· HP
· level
· Move(){ 저렇게 }
· Attack(){ BBB }
Minion

이렇게!


부모
· HP
· Level
· virtual Move()
· virtual Attack()
GameObject
자식
· HP
· Mana
· Level
· Move() override
· Skill()
· Attack() override
Champion
자식
· HP
· Level
· Move() override
· Attack() override
Minion

생성자, 소멸자 순서

```
#include <iostream>
using namespace std;
class Parent
{
private:
 int test1;
public:
 Parent() : test1(1)
 {
  cout << "부모 생성" << endl;
 }
 ~Parent()
 {
  cout << "부모 소멸" << endl;
 }
};
class Child : public Parent
{
private:
 int test2;
public:
 Child():test2(1)
 {
  cout << "자식 생성 " << endl;
 }
 ~Child()
 {
  cout << "자식 소멸 " <<endl;
 }
};
int main()
{
 Child *myobj = new Child;

 delete myobj;
 return 0;
}
```

실행 결과

부모 생성
자식 생성
자식 소멸
부모 소멸

```
#include <iostream>
using namespace std;
class Parent
{
private:
 int test1;
public:
 Parent() : test1(1)
 {
  cout << "부모 생성" << endl;
 }
 ~Parent()
 {
  cout << "부모 소멸" << endl;
 }
};
class Child : public Parent
{
private:
 int test2;
public:
 Child():test2(1)
 {
  cout << "자식 생성 " << endl;
 }
 ~Child()
 {
  cout << "자식 소멸 " <<endl;
 }
};
int main()
{
 Child *myobj = new Child;

 delete myobj;
 return 0;
}
```

소멸자에 virtual을 쓰는 이유

Main 함수에서
Parent *myobj = new Child;
delete myobj;
라고 선언 했을 경우,

실행결과

부모 생성
자식 생성
부모 소멸

부모 클래스의 포인터로
자식 클래스를 제어하는 경우
자식 클래스의 소멸자가
호출되지 않는 문제가 발생

즉 캐스팅 할 시에 문제 발생

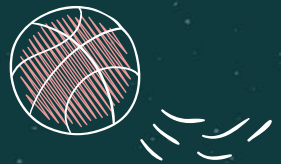
부모 클래스의 소멸자에
virtual 키워드를 선언

실행결과

부모 생성
자식 생성
자식 소멸
부모 소멸

캐스팅

03.



캐스팅이란?

자료형 혹은 포인터 간의 형 변환

일반적인 형변환

Ex)

```
double A;
int B = int(A);
```

static_cast

static_cast<자료형 or 포인터>(변수명)
일반적인 캐스팅과 가장 유사하다.

Ex)
```
double A;
int B = static_cast<int>(A)
```

struct 타입을 int나 double 타입으로 변환 할 수 없고,
float 타입을 포인터 타입으로 할 수 없다.
즉, 논리적으로 변환 가능한 타입만 변환 하며
변환이 안될 시 컴파일 에러를 통해 알려준다.

상속 관계에서 객체를 업, 다운 캐스팅 할 때,
사용할 수는 있지만, 다운 캐스팅 시 에러가 날 수도 있다.

dynamic_cast

dynamic_cast<자료형 or 포인터>(변수명)
계층 구조에 있는 서로 다른 클래스 혹은
부모 클래스에서 자식클래스로 다운 캐스팅시 사용

Ex)
```
Parent* p = new Child;
Child* c;
c = dynamic_cast<Child*>(p);
```

다운 캐스팅 시 안전하게 사용할 수 있다.
만약 캐스팅이 안될 시 NULL을 반환한다.
부모 클래스에 virtual 함수가 존재해야만 가능하다.

const_cast

const_cast<포인터>(변수명)
포인터 const 속성을 제거할 때 사용한다.

Ex)
```
const char* str;
char* removeConstStr = const_cast<char*>(str);
removeConstStr = nullptr; // 값 변경 가능
```

STUDY HARD!

이건 쓸일이 없는게 좋다!

reinterpret_cast

reinterpret_cast<자료형 or 포인터>(변수명)
강제로 형 변환을 수행한다.
정수형을 포인터형으로, 다른 클래스의 포인터 다 가능

Ex)
```
Parent* p = new Parent();
Unknown* x = reinterpret_cast<Unknown*>(p);
```

강제로 형 변환을 하기 때문에 논리적으로 불가능한 경우
런타임 에러가 발생하기 때문에 조심히 사용해야 한다.
변환 관계에 놓인 두 개체의 관계가 명확 하거나
특정 목적을 달성하기 위할 때만 사용하는 것이 좋다.

따라서 정리!

* 허용 되는 일반적인 변환을 하고 싶다?
  Ex) int - double -> static_cast: 같은 클래스

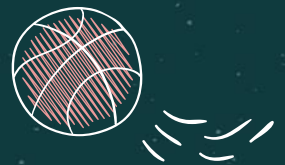
* 계층에 있는 서로 다른 클래스 객체의
  포인터, 참조 간의 변화? -> dynamic_cast

* const 속성을 제거 하고 싶다? -> const_cast

* 전혀 관계없는 두 포인터 혹은 참조 간 변화는?
  -> reinterpret_cast // 책임은 알아서^^

싱글톤(디자인 패턴)

04.



디자인 패턴 이란?

객체지향 프로그래밍 설계를 할 때 자주 발생하는 문제들을
피하기 위해 사용 되는 패턴.

많은 실무 프로그래머들이 인정한
효율적인 코딩 방법 or 구조

즉, 모두로 부터 인정된 합법적이고 똑똑한 얌생이 방법

싱글톤이란?

클래스가 오직 하나의 인스턴스(실체)를 갖는다!

그리고 그 하나의 실체를 전역적으로 공유할 수 있다!

따라서, 필요한 곳에 다 가져다 쓸수 있다!

예를 들어..

Mine Sweeper -> Screen Class

Screen은 하나 밖에 존재하지 않고,

다른 Class에서 많은 필요로 한다.

따라서 싱글톤으로 구현하기에 적합하다.

+ Input Class

싱글톤 구현(1)

```
1  class Singleton
2  {
3  private:
4      Singleton();
5      ~Singleton();
```

STUDY HARD!

생성자 함수를 private으로 선언한다.

외부에서 new 를 통한 instance 생성을 할 수 없게 만든다.

싱글톤 구현(2)

```
1  class Singleton
2  {
3  private:
4      Singleton();
5      ~Singleton();
6      static Singleton* Instance;
```

STUDY HARD!

인스턴스(실체)를 담을 수 있는 자기자신의 포인터 변수 선언

static으로 선언한다. (즉, 이 아이는 유일무이한 존재가 됨) +x÷

싱글톤 구현(3)

```
1  class Singleton
2  {
3  private:
4      Singleton();
5      ~Singleton();
6      static Singleton* Instance;
7
8  public:
9      static Singleton* GetInstance()
10     {
11         if (Instance == nullptr)
12         {
13             Instance = new Singleton;
14         }
15         return Instance;
16     }
```

인스턴스를 생성할 수 있는 static 함수 선언.

외부에서 사용해야 함으로 public으로 선언.

싱글톤 구현(4)

```
1  class Singleton
2  {
3  private:
4      Singleton();
5      ~Singleton();
6      static Singleton* Instance;
7
8  public:
9      static Singleton* GetInstance()
10     {
11         if (Instance == nullptr)
12         {
13             Instance = new Singleton;
14         }
15         return Instance;
16     }
17 };
18 Singleton* Singleton::Instance = nullptr;
```

Instance static 변수 초기화.

초기에 아직 안 만들어 졌다는 의미.

그리고
필요한 곳에 가져다
쓰면 된다!

혹시나 해서..

```
24  class Game
25  {
26  private:
27      Singleton* singleton;
28  public:
29      Game()
30          : singleton(Singleton::GetInstance())
31      {
32          // singleton = Singleton::GetInstance();
33      }
34      ~Game();
35  };
```

STUDY HARD!

Class 포인터 변수 선언 후
변수에 GetInstance를 호출하여 Instance를 넣어준다!
만약 Instance가 없다면 생성되고, 있다면 그걸 가져온다!

QnA

Discussion