Chapter 08 - 객체지향을 돕는 기능들

생성일	@2025년 11월 20일 오후 2:14
태그	

01) 컴포지션과 어그리게이션

다중 상속

- 부모 클래스를 여러 개 상속받아 자식 클래스를 정의
- 다양한 부모 클래스를 상속받기 때문에 많은 부분이 이미 정의
- 부모 클래스에 정의된 안정적이고 견고한 디자인 패턴을 그대로 활용
  → 개발 방법론이나 구조를 흔들림 없이 빠르게 전파할 수 있음

```
#include <iostream>
using namespace std;

// 캐릭터 글래스
class character {
public:
    character() : hp(100), power(100) {};
protected:
    int hp;
    int power;
};

// 캐릭터를 상속받는 플레이어 클래스
class player: public character {
public:
    player() { };
};
```

```
// 기본 몬스터 클래스
class monster {
public:
    monster() { };
    void get_damage(int _damage) { };
    void attack(player target_player) ( );
    void attack_special(player target_player);
};

void monster::attack_special(player target_player) {
    cout << "기본 공격 : 데미지 - 10 hp" << endl;
}
// 캐릭터와 기본 몬스터를 상속받는 몬스터 A
class monster_a : public monster, character {
public:
    void attack_special(player target_player);
};
```

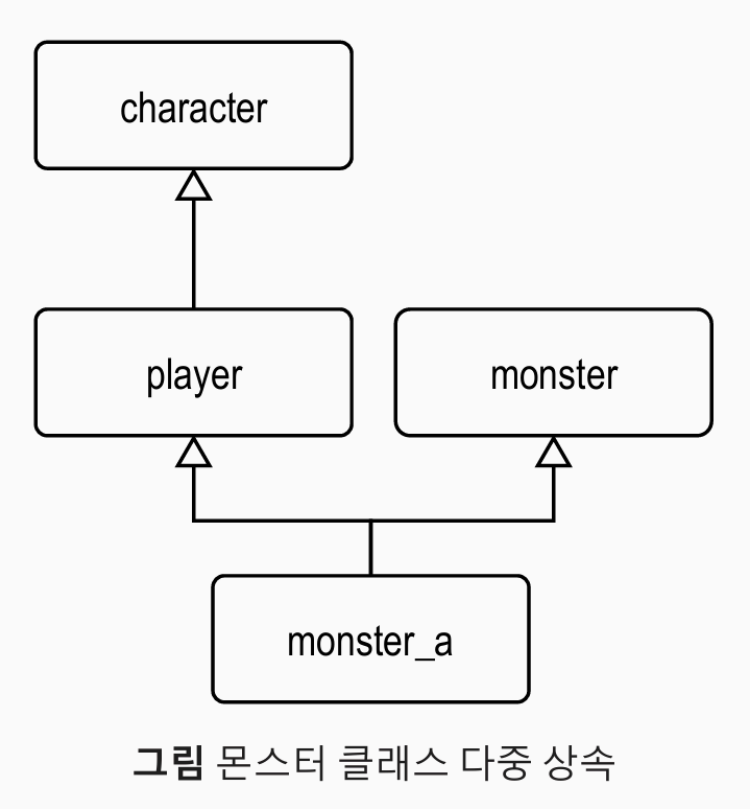

그림 몬스터 클래스 다중 상속

다중 상속 단점

- 클래스가 다양한 역할을 수행하게 되는 거대 클래스는 지양
  - 속성과 기능이 많아 사용하기가 어렵고, 부모 클래스가 변경되면 상속받은 모든 클래스에 영향
  - 변경 사항이 여러 곳에 영향을 주므로 바람직하지 않음
- 상속이나 사용 관계로 의존도가 높아지면 결합도가 높아짐
  - 결합도가 높으면 적은 수정에도 많은 수의 파일 컴파일이 진행됨

다중 상속 대안

- '컴포지션'과 '어그리게이션'
  - 멤버 변수로 포함하여 클래스를 재사용
    → 코드를 직접 사용하지 않아 변경 시 재 컴파일 없음
  - 클래스는 단일 속성과 기능을 가지므로 결합도는 낮아지고, 변경에 따른 영향이 적어짐
  - 컴포지션은 **포함 (part-of)**, 어그리게이션은 **사용 (has-a)**의 개념

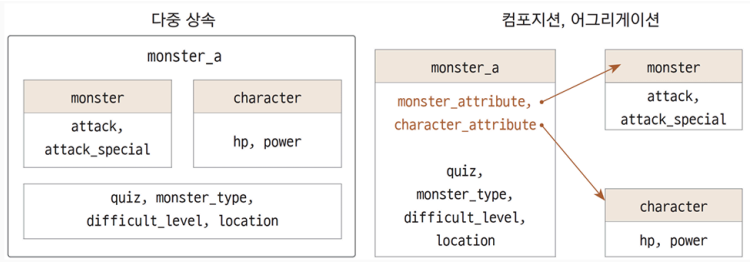

그림 다중 상속 vs 컴포지션, 어그리게이션

다중 상속 대안

- '컴포지션'과 '어그리게이션'
  - 멤버 변수로 포함
    - 재사용할 속성과 기능을 별도로 분리하여 해당 객체를 멤버 변수로 포함 (part-of), 포함한 클래스에 종속됨
    - 생명주기를 포함한 클래스가 관리하기 때문에 두 클래스의 생명주기가 동일함

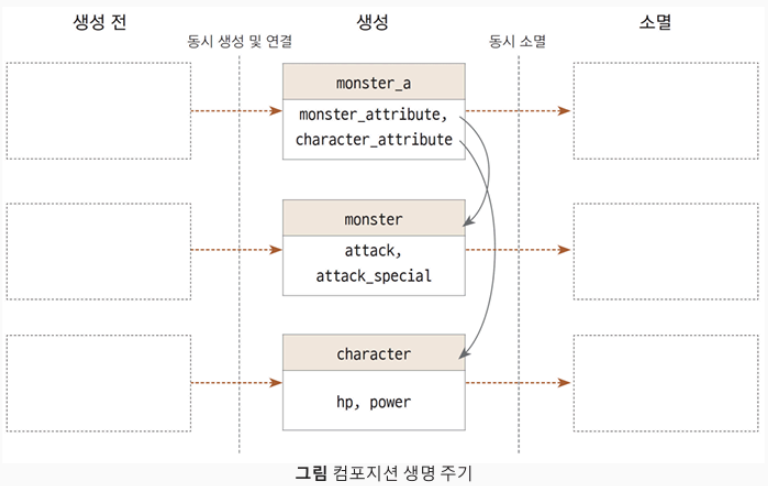

그림 컴포지션 생명 주기

8

어그리게이션

- 어그리게이션도 상속이 아니라 멤버 변수로 포함
  - 분리된 클래스의 객체를 포인터나 레퍼런스 변수로 포함. 분리된 클래스를 가지고 ('has-a') 사용
  - 클래스와 유연한 관계로 생명주기가 다름. 클래스를 직접 참조하거나, 상속받은 자식 클래스를 참조

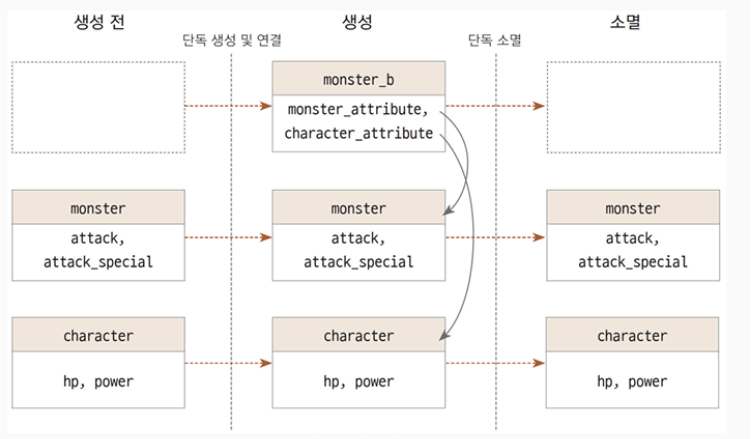

그림 컴포지션 생명 주기

9

```
// 생략
class monster {
public:
    monster() { };
    void get_damage(int _damage) { };
```

```
    virtual void attack(player target_player) {
        cout << "공격 : 데미지 - 10 hp" << endl;
    };
};

class monster_2nd_gen : public monster {
public:
    virtual void attack(player target_player) override {
        cout << "새로운 공격 : 데미지 - 30 hp" << endl;
    };
};

class monster_a {
public :
    void attack(player target_player) {
        main_role.attack(target_player);
    };
private:
    // 캐릭터와 몬스터 객체를 직접 생성 ( 몬스터 A와 생명주기가 같
음 )
    character main_body;
    monster main_role;
};

class monster_b {
public:
    // 레퍼런스 멤버 변수는 초기화 목록으로 초기화
    monster_b(character &ref_character, monster &ref_monste
r)
        : main_body(ref_character), main_role(ref_monster)
{ };
    void attack(player target_player) {
    main_role.attack(target_player);
};
private:
    // 캐릭터와 몬스터 객체를 참고 (몬스터 B와 생명 주기가 다름)
    character &main_body;
    monster &main_role;
```

```
};

int main() {
    player player_1;
    character character_obj;
    monster monster_obj;
    monster_2nd_gen monster_new_obj;

    // 내부에서 객체를 직접 생성
    monster_a forest_monster;

    // 외부 객체 전달
    monster_b tutorial_monster(character_obj, monster_obj);
    monster_b urban_monster(character_obj, monster_new_ob
j);

    cout << "몬스처 A 공격" << endl;
    forest_monster.attack(player_1);

    cout << endl << "몬스터 A 공격" << endl;
    forest_monster.attack(player_1);

    cout << endl << "몬스터 B 공격" << endl;
    tutorial_monster.attack(player_1);
    urban_monster.attack(player_1);

    return 0;
}
```

실행 결과

```
몬스터 A 공격
공격 : 데미지 - 10 hp

몬스터 B 공격
공격 : 데미지 - 10 hp
새로운 공격 : 데미지 - 30 hp
```

- 어그리게이션은 클래스 설계 이후에도 다양한 객체를 받아 들일 수 있다.

→ **늦은 바인딩** : **설계 이후** 구현체를 바꿔서 참조하게 하는 것 (활용도가 매우 높은 설계 방법)

02) 가상 함수와 동적 바인딩

가상 함수

- 멤버 함수 가운데 자식 클래스에서 오버라이딩(재정의)해야 하는 함수
  - (또는 오버라이딩 하기를 기대하는 함수)
- 일반 멤버 함수도 자식 클래스에서 오버라이딩 할 수 있음
  - 오버라이딩이 필요하다는 의미를 전달할 수 없음
    → 문법적으로 명확하지 않음
  - 부모 클래스로 업캐스팅 시 호출되는 함수가 다름
    → 다형성 구현의 원리
  - 함수 선언 가장 첫 부분에 virtual 키워드 추가
    - 부모, 자식 클래스 모두 virtual 키워드 추가해 주어야 함

virtual 반환_형식 함수_이름(매개변수);

가상 함수 선언

- 오버라이딩 하는 자식 클래스에서는 **override 키워드** 추가
  - **선언부**의 가장 마지막 부분

virtual 반환_형식 함수_이름(매개변수) override;

문법 요약

```
class monster {
public:
    // 1. 가상 함수 선언
```

```
    virtual void attack_special(player target_player);
};
// 2. 가상 함수 정의
void monster::attack_special(player target_player) { ...생
략... }

class monster_c : public monster {
public:
    // 3. 가상 함수 오버라이드 선언
    virtual void attack_special(player target_player) ovrri
de;
};
// 4. 가상 함수 오버라이드 선언
void monster_c::attack_special(player target_player) { ...
생략... };
```

1. **가상 함수 선언** : 가상 함수 선언은 앞 부분에 virtual 키워드를 추가하는 것만 다르고 일반 함수 선언과 같다.
2. **가상 함수 정의** : 가상 함수 정의는 일반 함수와 같다.
3. **가상 함수 오버라이드 선언** : 자식 클래스에서 가상 함수 오버라이드를 선언할 때는 마지막 부분에 override 키워드를 추가한다.
4. **가상 함수 오버라이딩 (재정의)** : 자식 클래스에 오버라이드 된 가상 함수 정의는 일반 오버라이딩 함수와 같다.

가상 함수로 다형성 구현

- 자식 클래스가 부모 클래스를 대체하더라도 **(업캐스팅)** 오버라이딩한 함수가 호출되어야 함.

```
#include <iostream>
using namespace std;

class character {
public:
    character() : hp(100), power(100) { };
```

```
protected:
    int hp;
    int power;
};

class player : public character {
public:
player() { };
};

class monster {
public:
    monster() { };
    void get_damage(int _damage) { };
    void attack(player target_player) { };
    virtual void attack_special(player target_player); //
가상 함수 선언
};

void monster::attack_special(player target_player) {
    cout << "기본 공격 : 데미지 - 10 hp" << endl;
}

class monster_a : public monster, character {
public:
    // 가상 함수 오버라이드 선언
    virtual void attack_special(player target_player) overr
ide;
};

// 가상 함수 오버라이딩
void monster_a::attack_special(player target_player) {
    cout << "인텡글 공격 : 데미지 - 15 hp" << endl;
}

...생략...

int main() {
```

```
    player player_1;

    monster_a forest_monster;

    monster &mon = forest_monster;
    monster_a &mon_a = forest_monster;

    cout << endl << "부모 클래스로 업캐스팅 후 공격" << endl;
    mon.attack_special(player_1);

    cout << endl << "자식 클래스로 공격" << endl;
    mon_a.attack_special(player_1);

    cout << endl << "범위 연산자로 공격" << endl;
    monster_a.monster::attack_special(player_1);

    return 0;
}
```

가상 함수로 다형성 구현

- **업캐스팅** 후 가상 함수 호출
  - monster_a 클래스 forest_monster 객체를 monster와 monster_a의 레퍼런스 변수에 각각 대입
  - monster에 대입된 forest_monster 객체는 **업캐스팅 됨**
    - 가상 함수가 아니고 범위 연산자가 없다면 부모 범위로 수행
  - **가상 함수로 정의**
    - 자식 클래스에서 **오버라이딩한 함수가 호출**됨

실행 결과

```
부모 클래스로 업캐스팅 후 공격
인텡글 공격 : 데미지 - 15 hp

자식 클래스로 공격
```

```
인텡글 공격 : 데미지 - 15 hp

범위 연산자로 공격
기본 공격 : 데미지 - 10 hp
```

가상 함수로 다형성 구현

- 일반 함수로 오버라이딩 했을 경우

```
#include <iostream>
using namespace std;
...(생략)...

class monster {
public:
    monster() { };
    void get_damage(int _damage) { };
    void attack(player target_player) { };
    void attack_special(player target_player);
};

void monster::attack_special(player monster_player) {
    cout << "기본 공격 : 데미지 - 10 hp" << endl;
}

class monster_a : public monster, character {
public:
    // 일반 함수 오버라이드 선언
    void attack_special(player target_player) override;
};

void monster_a::attack_special(player target_player) {
    cout << "인텡글 공격 : 데미지 - 15 hp" << endl;
}

...(생략)...
```

```
int main() {
    player plater_1;

    monster_a forest_monster;

    monster &mon = forest_monster;
    monster_a &mon_a = forest_monster; // 업캐스팅 발생

    cout << endl << "부모 클래스로 업캐스팅 후 공격" << endl;
    mon.attack_special(player_1); // monster 멤버 함수 호출

    cout << endl << "자식 클래스로 공격" << endl;
    mon_a.attack_special(player_1);

    return 0;
}
```

함수의 동작 바인딩

- 가상 함수 호출의 원리
  - 가상 함수 테이블을 활용해 동적으로 바인딩
- 바인딩
  - 함수 호출이나 변수 참조가 동작 코드와 연결되는 과정
    - 프로그램 실행은 메모리를 이동하면서 실행이 되는데, 실행 코드 메모리를 연결하는 과정임
  - 바인딩은 정적 바인딩과 동적 바인딩 두 가지 종류가 있음
    - 정적 바인딩 : 이른 바인딩
    - 동적 바인딩 : 늦은 바인딩
  - 정적 바인딩은 컴파일 시점에 동작
    - 정적으로 바인딩 되는 대상은 컴파일할 때 결정되어 프로그램이 실행되는 동안 유지
    - 일반 변수와 함수, 클래스, 정적 멤버 함수, 템플릿 등 대부분은 정적 바인딩

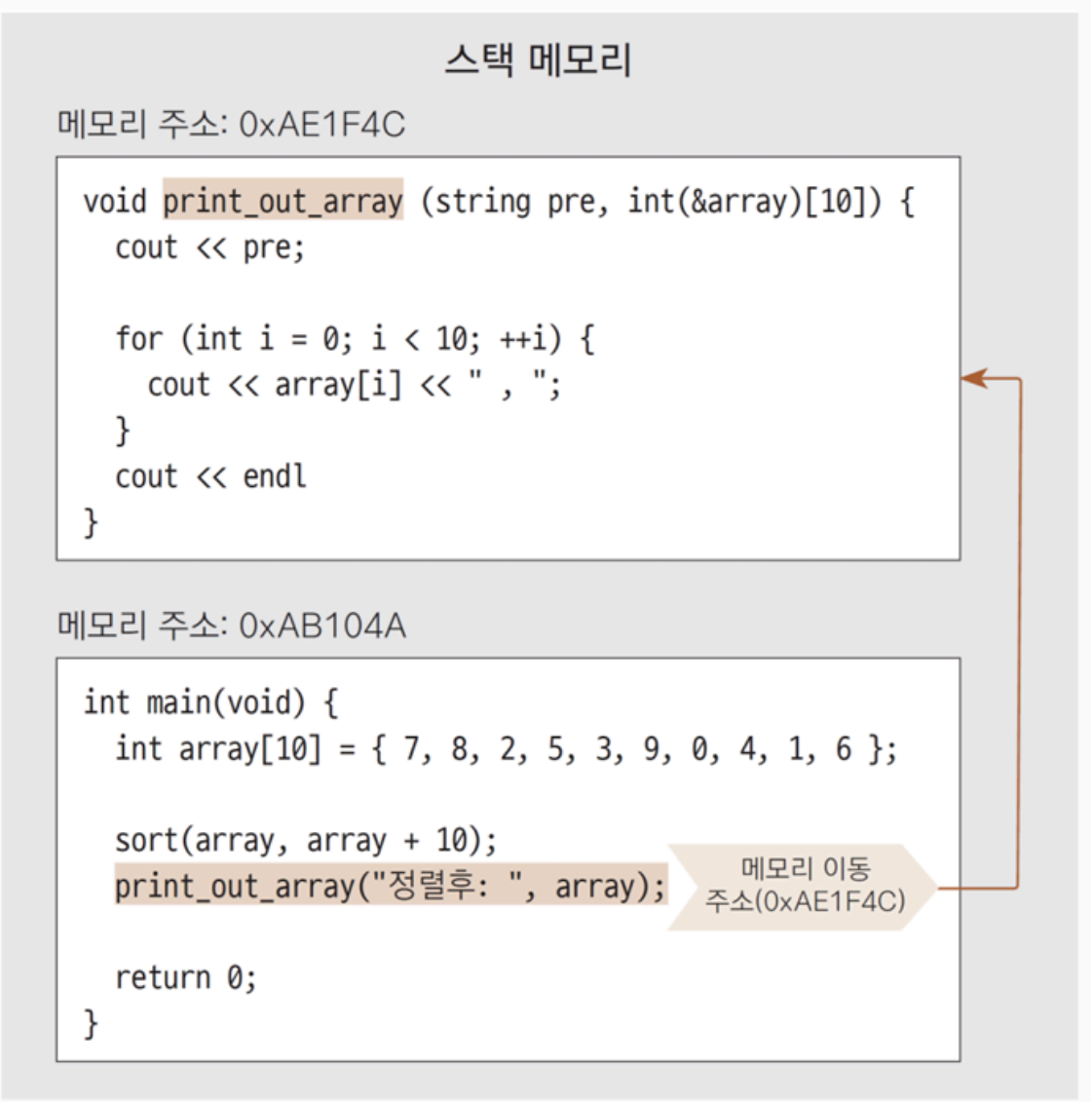

그림 함수 호출에서 바인딩

- 정적 바인딩에서 고정 되는 것
  - 정적 바인딩이라고 해서, 항상 같은 메모리 주소가 저장되는 것이 아니라, 바인딩 대상이 컴파일 시점에서 결정되는 것이다.

→ 실제 참조할 메모리 주소는 프로그램을 실행할 때마다 달라진다.

→ 정적 바인딩은 실제 메모리 주소를 고정하는 것이 아니라, 바인딩 대상이 있는 위치를 고정하는 것으로
이해하면 된다.

→ **연결의 대상이 컴파일 시점에 정해진다.**

함수의 동적 바인딩

```
#include <iostream>
using namespace std;

...(생략)...

class monster{
public:
    monster() { };
    virtual void attack_special(player target_player);
};
void monster::attack_special(player target_player) {
    cout << "기본 공격 : 데미지 - 10 hp" << endl;
}

class monster_a : public monster, character {
public:
    virtual void attack_special(player target_player) overr
ide;
};

void monster::attack_special(player target_player) {
    cout << "인텐글 공격 : 데미지 - 15 hp" << endl;
}

int main() {
    player player_1;
    monster monster_monster;
    monster_a forest_moster;

    monster_monster.attack_special(player_1);

    monster *mon = &forest_monster;
    cout << endl << "부모 클래스로 업캐스팅 후 공격" << endl;
    mon -> attack_special(player_1);

    mon = &mothre_monster;
```

```
    cout << endl << "부모 클래스로 공격" << endl;
    mon -> attack_special(player_1);

    return 0;
}
```

실행 결과

```
기본 공격 : 데미지 - 10 hp

부모 클래스로 업캐스팅 후 공격
인텐글 공격 : 데미지 - 15 hp

부모 클래스로 공격
기본 공격 : 데미지 - 10 hp
```

함수의 동적 바인딩

- **동적 바인딩**
  - 대상이 실행 시점에 결정되며, 변경될 수 있음
  - 가상 함수, 자식 클래스로 치환된 부모 클래스의 포인터가 동적 바인딩의 대표 예
- 동적으로 바인딩 된 대상은 프로그램이 실행되는 동안에 수시로 변경이 가능
- **정적 바인딩 vs 동적 바인딩**
  - 부모 클래스의 객체 mother_monster 멤버 함수 직접 호출

  → 정적 바인딩 함수의 주소로 바로 이동
  - 클래스 포인터 mon 호출 시 클래스의 객체로 이동 후 본 클래스함수의 주소로 이동

  → mon이 가리키는 객체의 attack_special 함수 호출 시 동적으로 바인딩 된 주소를 찾아서 호출

```
mother_monster.attack_special(player_1);
-> call    mother::attack_special (0xAE1FAC) (1)
mother *mon = &forest_monster;
-> lea     rax, [forest_monster]             (2)
-> mo       aword ptr [mon], rax             (3)
```

```
mon -> attack_specoal(player_1);
-> call     aword ptr [rax]                    (4)
```

1. monster::attack_special 함수를 주어진 메모리 주소 (0xAE1FAC)에서 호출
2. forest_monster 변수의 주소를 rax 레지스터*에 탑재 (lea)
3. 그 주솟값을 mon이라는 메모리 주소에 64비트 데이터로 복사 (mov)
4. rax 레지스터에 저장된 주소로 이동하여 해당 주소에 위치한 함수를 호출

- **동적 바인딩** 메모리 이동 단계

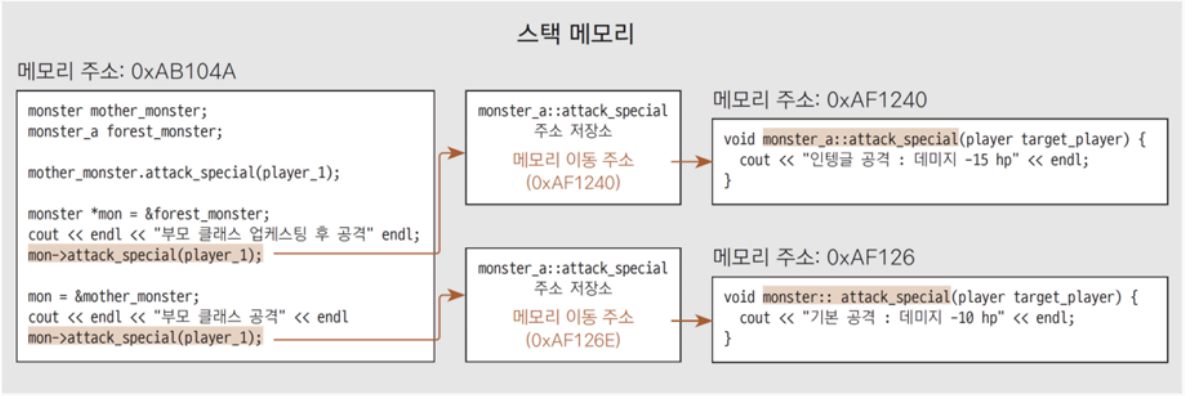

그림 함수의 동적 바인딩

가상 함수 테이블

- 클래스 계층 구조 최상위에 존재하는 **가상 함수에 대한 메모리 정보**
- **업캐스팅** 후에도 자식 테이블이 호출될 수 있도록 정보 제공하는 구조
- 가상 함수가 있는 객체를 생성하면, **가상 함수 테이블을 가리키는 포인터 (_vfptr)** 생성
  - 가상 함수를 호출 시 _vfptr을 활용, **가상 함수의 개수**에 따라 **_vfptr 크기가 변경**됨

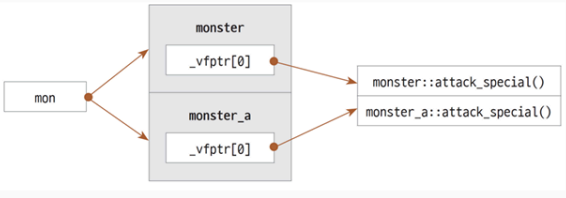

그림 가상 함수 테이블 구조

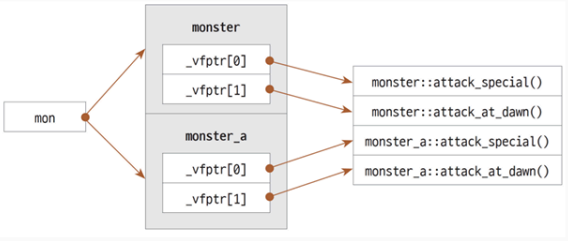

그림 가상 함수가 여러 개일 때 가상 함수 테이블 구조

- **비주얼 스튜디오에서 `_vfptr` 확인하기**
  - 비주얼 스튜디오 편집창에서 49번 줄에 F9를 눌러 중단점을 설정
  - **F5**를 눌러 디버깅을 시작, 49번 줄에서 실행이 멈춤
  - **F10**을 누르면 한 줄씩 나타나면서 디버깅이 진행됨
  - 노랑 화살표가 나타나 다음에 실행할 줄을 가르킴, **52번 줄까지 코드**를 실행
  - 하단의 **[자동]**이라는 창에서 mon과 forest_monster를 펼쳐 확인

순수 가상 함수

- 부모 클래스에서는 가상 함수를 **정의 없이 선언**만 진행
- **자식 클래스에서 반드시 재정의** 해야 한다는 것을 문법적으로 알림
- 순수 가상 함수로 **명시적으로** 남겨둘 수도 있음 (순수 가상 함수로 다시 선언)
- 순수 가상 함수 선언

```
virtual void attack_special(player target_player) = 0;
```

```
#include <iostream>
#include <list>

using namespace std;

class character {
public:
    character() : hp(100), power(100) { };
    void get_damage(int _damage) { };
protected:
    int hp;
    int power;
};
class player : public character {
public:
    player() { };
```

```
};

class monster {
public:
    monster() { };
    void attack(player target_player) { };
    virtual void attack_special(player target_player);
    virtual void attack_at_dawn() = 0;
};

class monster_a : public monster, character {
public:
    // 상속 받은 함수 오버라이드 선언
    virtual void attack_special(player target_payer) overri
de;
    virtual void attack_at_dawn() override;
};

// 순수 가상 함수 오버라이딩
void monster_a::attack_at_dawn() {
    cout << "동쪽에서 기습!" << endl;
}

void monster_a::attack_special(player target_player) {
    cout << "인텡글 공격 : 데미지 - 15 hp" << endl;
}

class monster_b : public monster, character {
public:
    virtual void attack_special(player target_player) overr
ide;
    virtual void attack_at_dawn() override;
};

void monster_b::attack_at_dawn() {
    cout << "적진에 조용히 침투하여 방화!" << endl;
}
```

```
void monster_c : public monster, character {
public:
    virtual void attack_special(player target_player) overr
ide;
    virtual void attack_at_dawn() override;
};

void monster_c::attack_special(player target_player) {
    cout << "강력 뇌전 공격 : 데미지 - 100 hp" << endl;
}

void monster_c::attack_at_dawn() {
    cout << "남쪽에서 적진을 향해 대포 발사!" << endl;
}

int main() {
    list<monster*> mon_list;

    monster_a first_monster;
    mon_list.push_back(&first_monster);

    monster_b second_monster;
    mon_list.push_back(&second_monster);

    monster_c third_monster;
    mon_list,push_back(&third_monster);

    for (auto item : mon_list {
        item -> attack_at_dawn(); // 리스코프 치환 원칙 적용 -
다형성 적용
        }
        return 0;
    }
```

실행 결과

```
동쪽에서 기습!
적진에 조용히 침투하여 방화!
```

```
남쪽에서 적진을 향해 대포 발사!
```

- **attack_at_dawn( )**
  - monster_a, monster_b, monster_c에서 오버라이딩
  - 리스코프 치환 원칙이 적용됨
  - **오버라이딩 하지 않으면 문법적 오류 발생** (→ **순수 가상함수**)
- **attack_special( )**
  - 자식 클래스에서 오버라이딩 하지 않으면 부모 클래스의 정의를 그대로 사용함
  - 오버라이딩 하지 않아도 문법 오류 발생하지 않음

    → **순수 가상함수가 아님** (→ **가상함수**)

가상 소멸자

- **업캐스팅 된 객체의 소멸 시 문제 발생**
  - 소멸자는 메모리 해제 등 리소스 정리를 위해서 사용 됨
  - 업캐스팅된 상태의 객체 메모리가 해제될 때 소멸자 호출?
    - **업캐스팅 되지 않은 객체**는 객체에서 정의한 **소멸자, 부모 클래스의 소멸자 순**으로 호출됨
  - **업캐스팅 된 경우** 업캐스팅 된 클래스의 범위에서 **소멸자가 호출**됨
    - 업캐스팅된 **객체의 소멸자는 호출 되지 않음**

```
#include <iostream>
using namespace std;

class monster {
public:
    monster();  // 생성자
    ~monster();  // 소멸자
private:
    int *dummy;
};
```

```
monster::monster() {
    cout << "~monster() 생성자 호출" << endl;
    delete dummy;  // 메모리 해제
}

class monster_a : public monster {
public:
    monster_a();
    ~monster_a();
private:
    int *dummy_a;
};

monster_a::monster_a() {
    cout << "monster_a() 생성자 호출" << endl;
    dummy_a = new int;
}

monster_a:::~monster_a() {
    cout << "~monster_a() 소멸자 호출" << endl;
    delete dummy_a;
}

int main() {
    monster_a *mon = new monster_a();

    delete mon;
    return 0;
}
```

실행 결과

```
monster() 생성자 호출
monster_a() 생성자 호출
~monster_a() 소멸자 호출
~monster() 소멸자 호출
```

- **monster::~monster() 호출**

    - 생성된 객체와 무관
    - **업캐스팅 된 클래스 범위의 소멸자가 호출**
- **monster_a 내 동적 할당된 멤버 변수**
    - 소멸자가 호출되지 않아 **메모리 누수** 발생

가상 소멸자

- **자식 클래스의 가상 소멸자부터** 호출
    - 업캐스팅되지 않을 때와 동일한 순서로 호출

```
#include <iostream>
using namespace std;

class monster() {
public:
    monster():
    virtual ~monster();
private:
    int *dummy;
};
... (생략) ...
int main() {
    monster *mon = new monster_a();  // 부모 클래스로 업캐스팅
    delete mon;
    return 0;
}
```

실행 결과

```
monster() 생성자 호출
monster_a() 생성자 호출
~monster_a() 소멸자 호출
~monster() 소멸자 호출
```

03) 추상 클래스와 정적 멤버

추상 클래스

- 구체화 되지 않은 **추상적인 클래스**
  - **순수 가상 함수를 포함**하고 있어서 구체화되지 않았다는 의미
  - 일반 멤버 변수나 멤버 변수를 **포함 여부는 상관 없음**
  - 다형성 구현 및 다양한 디자인 패턴에 활용됨

- **구상 클래스**
  - 추상 클래스와 대비되는 **구체화된 클래스**, 지금까지 사용했던 일반 클래스

- 객체 선언이 불가능한 **추상 클래스**
  - 구체화되어 있지 않기 때문에 **객체를 선언할 수가 없음**
  - 추상 클래스를 상속 받아 **순수 가상 함수를 오버 라이딩**한 클래스로 객체를 선언

```
#include <iostream>
#include <list>
using namespace std;
...(생략)...

// 추상 클래스
class monster {
public:
    monster();  // 생성자
    virtual ~monster();  // 소멸자
    virtual void find_route() = 0;
    virtual void attack_special(player target_player) = 0;
// 순수 가상함수
};
...(생략)...

class monster_a : public monster {
```

```
public:
    // 순수 가상함수 오버라이드 선언
    virtual void attack_special(player target_player) overr
ide;
    virtual void find_route() override; // 순수 가상함수 오버
라이드 선언
};

// 추상 클래스의 순수 가상 함수 구현
void monster_a::attack_special(player target_player) {
    cout << "인텡글 공격 : 데미지 - 15 hp" << endl;
}

// 추상 클래스의 순수 가상 함수 구현
void monster_a::find_route() {
    cout << "깊이 우선 탐색 (DFS)" << endl;
}

// 전역함수
void monster_routine(monster *mon, player target_player) {
    mon -> find_route();
    mon -> attack_special(target_player);
}

int main() {
    list<monster*> mon_list;
    monster_a first_mon;
    monster_b second_mon;
    monster_c third_mon;
    player target_player;

    mon_list.push_back(&first_mon);
    mon_list.push_back(&second_mon);
    mon_list.push_back(&third_mon);

    for (auto mon : mon_list) {
        monster_routine(mon, target_player);
    }
```

```
    return 0;
}
```

실행 결과

```
...(부모 클래스 생성자 출력 생략)...
깊이 우선 탐색(DFS)
인텡글 공격 : 데미지 - 15 hp
너비 우선 탐색(BFS)
가상 공격 : 데미지 - 0 hp
다익스트라 최단 경로 알고리즘
강력 뇌전 공격 : 데미지 - 100 hp
...(부모 클래스 소멸자 출력 생략)...
```

- **전략 패턴**
  - 앞 페이지 예제는 전략 패턴의 한 형태
  - 전역 monster_routine 매개변수로 전달받은 부모 클래스의 포인터로 find_route와 attack_special 함수 호출
    - 업캐스팅 된 부모 클래스의 (순수) 가상 함수로 다른 구현체를 사용함
  - 동일한 흐름에 다른 알고리즘을 사용하고 싶을 때 흐름과 알고리즘을 분리
    - 흐름을 사용하는 곳에서는 알고리즘이 어떻게 구현되었는지 신경 쓰지 않아도 됨
  - 다른 알고리즘으로 쉽게 교체할 수도 있음
    - 소프트웨어의 유지, 보수성을 높이고 새로운 알고리즘을 추가하기도 쉬워짐

정적 멤버

- '정적 멤버 변수', '정적 멤버 변수'
  - 멤버 선언 앞에 static 키워드를 사용한 정적 멤버, 객체 선언 없이 바로 사용할 수 있음
  - 정적 멤버는 클래스 키워드에 속하지만 특정 인스턴스에 종속되지 않음

- 정적 멤버 사용
  - 범위 연산자(::)를 사용하여 접근
  - 객체가 선언되었을 경우 일반 멤버처럼 이용 가능

```
class_name::static_variable = 10;
class_name::static_function();
```

- 접근 제어
  - 정적 변수와 다르게 접근 지정자에 따른 접근 제어가 됨
  - private로 선언 되었을 경우 클래스 외부에서 접근이 불함

- 정적 멤버 변수
  - 범위 지정자가 적용된 정적 변수; 같은 클래스에서 만들어진 객체에서 공유하는 값
  - 또는 외부에서 참고할 수 있는 동일 클래스로 선언된 객체들의 값

- **정적 멤버 함수**
  - 클래스의 객체를 클래스 내부에서 직접 관리하고 싶을 때, 또는 유틸리티 함수를 만들 때 활용
  - **유틸리티 함수**란, 문자열 조작, 날짜나 시간 계산, 수학적 계산 등 특정 작업을 지원하는 함수
    - 예) 행렬, 벡터 연산을 하는 선형대수의 연산자 정의 시 :

      연산자들은 행렬, 벡터 만을 매개변수로 사용,

      범위 연산자로 선형대수와 관련된 함수임을 명시적으로 표현

      행렬, 벡터 연산이 필요할 때마다 객체를 생성하는 번거로움이 없음
    - 객체를 클래스 내부에서 직접 관리하는 예시 → **팩토리 패턴**
      - 예) 몬스터 객체를 생성하는 클래스 멤버를 정적 멤버 함수로 정의