[2019학년도 봄학기 컴퓨터프로그래밍]

Project_1 보고서

생활과학대학 소비자아동학부 소비자학전공 2015-15356 이준희

(* extra-credit을 위한 코드를 작성했습니다.)

1. Development/Implementation Environment

Visual Studio Code (code runner 확장기능 설치)

버전: 1.33.1 (user setup)
커밋: 51b0b28134d51361cf996d2f0a1c698247aeabd8
날짜: 2019-04-11T08:27:14.102Z
Electron: 3.1.6
Chrome: 66.0.3359.181
Node.js: 10.2.0
V8: 6.6.346.32
OS: Windows_NT x64 10.0.17763

2. Specific code description

```
#include <iostream>
#include <cstring>
#include <vector>
#include <cstdlib>
#include <ctime>
```

- 사용한 라이브러리 목록

① input 처리

main함수에서 test case 번호를 입력받은 후, 입력받은 번호에 해당하는 string을 char[] 형태로 받아와 to_journey_list 함수의 인자로 전달한다. 그리고 to_journey_list 함수의 반환값을 to_env_list 함수의 인자로 전달한다. 그렇게 to_env_list에서 최종적으로 반환되는 값을 env_list 벡터에 할당한다.

```
vector<Environment*> env_list = to_env_list(to_journey_list(chosen_TC));
```

```
Journey* my_journey = new Journey(env_list, text_graphic_initializer(env_list), mode_num);
```

또한, 추후 Journey class를 만들어줄 때, env_list를 인자로 받는 함수 text_graphic_initializer의 값이 constructor 인자에 들어가게 된다. 이렇게, input 으로부터 필요한 정보를 추출해 데이터를 가공하는 작업을 위해 to_journey_list, to_env_list, text_graphic_initializer, 총 세 가지 helper function을 정의해주었다.

```
////////////////////////////
// helper function prototypes
vector<char*> to_journey_list(char TC[]);
vector<Environment*> to_env_list(vector<char*> whole_journey);
string text_graphic_initializer(vector<Environment*>& env_list);
```

세 helper function의 프로토타입은 위와 같다.
to_journey_list는 테스트케이스 문자열을 char[]형태로 받아, 각 environment 별로 나눠서 char* 벡터에 값들을 저장한다. to_env_list는 그렇게 parsing된 char* element를 담은 vector를 인자로 받아 각각 Environment 객체들을 생성해 Environment*를 element로 가지는 vector를 반환한다(이는 추후 Journey class에서 주요하게 활용된다). text_graphic_initializer는 위 두함수와는 다소 별개의 것으로 볼 수 있는데, extra credit 부분 처리를 위해 Journey class를 만들기 이전 environment vector를 읽어들여 그에 대응하는 text_graphic을 string형태로 생성해 반환해주는 함수이다.

② main함수 주요부분

```
Journey* my_journey = new Journey(env_list, text_graphic_initializer(env_list), mode_num);
my_journey->initial_status_print();
my_journey->change_env();

int move_num;
while (!(my_journey-> journey_finish)){
    if (!skip_ask){
        cout<<"Next Move? (1,2)"<<endl;
        cout<<"CP-2015-15356>";
        cin>> move_num;
        //cin.ignore();
    }
    skip_ask = false;
    my_journey-> move(move_num);
}
```

위에서와 같이 인풋값 처리를 거치고 추출한 데이터들을 활용해, Journey class를 만들어준다. Journey class가 만들어지면 initial status를 print하며, 본격적으로 journey를 시작하기 위해 첫 번째 environment로 시작이 이루어질 수 있게 change_env() 함수를 실행해준다. 그 뒤엔, journey가 끝날 때까지 Next Move 여부를 물으며, 입력받은 mode number에 해당하는 move 메소드를 실행시켜준다. (단, Next Move?를 묻고 프롬프트를 띄워 인풋을 받는 행위는, skip_ask = true가 되는 순간에는 실행되지 않는다. 이때는, unexpected environment를 만나 한 싸이클을 내부적으로 처리해줘야 하는 단계이다.

이렇게 생성한 Journey가 끝나 루프를 벗어나게 되면, 그 바깥에 또 한겹 존재하는 while loop로 인해 다시 새로운 테스트케이스 번호를 받고, 0이 입력되지 않는 이상 계속해서 위와 동일한 과정을 반복하게 된다.

③ global variables

```
#include <iostream>
#include <cstring>
#include <vector>
#include <cstdlib>
#include <ctime>

using namespace std;

bool global_unexpected = false;
bool skip_ask = false;
```

global variable을 자주, 중점적으로 사용하는 것은 바람직하지 않지만, 코드의 간결성을 위해 편의상 global_unexpected, skip_ask 이렇게 두 개의 bool형 global variable을 선언해 주었다. 이 둘을 선언한 이유는 무엇보다 unexpected environment를 만났을 때 모든 처리를 implicit하게 내부에서 처리하기 위함이다.

구체적으로 살펴보면, skip_ask는 unexpected environment를 만났을 때, main 함수에서 Next Move?를 두 번 묻는 등의 문제를 발생시키면 안 되므로, skip_ask 조건이 충족 될 때는 한번 묻는 행위를 건너뛰고 내부적으로 그 다음 단계의 일을 수행하게 한다(skip_ask값은 Journey class 내 move함수에서, 또 main함수에서, 이렇게 총 두 곳에서만 조작이 가능하다). (global variable이기 때문에 값이 프로그램 내내 유기적으로 연결되고, 유지된다. 따라서 한 journey가 끝난 후 반드시 다시 false값을 할당해야 함에 유의한다.)

다음으로 global_unexpected는 현재 Journey가 unexpected를 만난 상태인지 아닌지를 나타내 주는 값으로, 조작은 오로지 Journey 클래스 내 move와 change_status 함수에서만 가해줄 수 있다. main함수 - Journey class (특히 그중에서도 move와 change_status 멤버함수) - vehicle derived class들의 멤버함수, 이렇게 세 레이어를 오가기 위해 만들어준 global 변수라고 할 수 있다 (원래는 각각 Journey class와 vehicle derived class들에 unexpected를 지나고 있는지 아닌지에 대한 정보를 데이터멤버로 넣어주고, get/set함수들을 더불어 선언해주었는데, 코드가 지나치게 복잡해지는 경향이 있어 다시 global variable하나를 선언하는 방향으로 수정했다.)

④ class 구성 (1) - Journey Class

Journey class가 main함수에서 가장 핵심적인 역할들을 수행하므로, 다양하게 정의해준 클래스들 중 가장 먼저 살펴보도록 하자.

```
class Journey{
    public:
    bool journey_finish;
    int graphic_index;
    int extra_mode_num;
    int move_mode;

    Journey(vector<Environment*> env_list, string text_graphic, int extra_mode_num){
        this-> extra_mode_num = extra_mode_num;
        this -> env_list = env_list;
        this -> text_graphic = text_graphic;
        total_energy = 1000;
        total_oxygen = 100;
        journey_finish = false;
        total_distance=0;
        graphic_index=1;

        current_env = new Environment(0,0); // dummy data
        car_charger_broken = false;
        last_loop = false;
    }
```

```
private:
bool gotta_change_env=false;
bool last_loop;
bool car_charger_broken;
int vector_count=0;
vector<Environment*> env_list;
int total_distance;
int total_energy;
int total_oxygen;
vector <string> mode_change;
vector <int> energy_change;
vector <int> oxygen_change;
vector <int> speed_change;
Environment* current_env;
Vehicle* current_vehicle;
string text_graphic;
```

Journey class의 data member들을 부분적으로 살펴보면 위와 같다.
journey_finish는 본 Journey가 최종적으로 아예 끝난 상태인가를 나타내는 boolean 값이고, graphic_index는 graphic에서 현재 vehicle이 어디에 위치하는지, 그 위치정보 index를 담고 있는 int값이다. extra_mode_num은 프로그램이 시작할 때 맨 처음 input으로 받았던 모드값이 EXTRA모드(값 1)인지, NORMAL 모드(값 2) 인지를 담고 있다.

다음으로 private member들을 살펴보면, gotta_change_env는 다음에 move함수가 행해질 때 change_env를 실행해야 하는지 아닌지에 대한 정보를 지니고 있는 boolean값이고, last_loop은 현재 가장 마지막 environment를 처리하고 있을 경우 true, 아닐 경우 false가 되는 boolean값이며, car_charger_broken은 이번 journey에서 사용할 수 있는 car solar panel energy charger가 망가진 상태인지 아닌지에 대한 정보를 담고 있는 boolean값이다. 또한, vector_count는 Environment 벡터 내 객체들에 대하여 루프를 돌기위해 기록해놓는 index 값이다 (따라서 0으로 초기화해준다).

env_list 벡터는 Journey constructor에서 인자로 받아온 Environment 객체들로 이루어진 벡터 데이터이다. total_distance는 본 Journey에서 현재까지 거쳐 온 거리 정보를 담고 있으며, total_energy는 현재까지의 총 에너지 값, total_oxygen은 현재까지의 oxygen rate을 나타낸다. 다음으로 mode_change, energy_change, oxygen_change, speed_change는 Journey를 끝낼 때 print해줄 Black Box 내용을 위해 각 journey 말미에 mode/ energy/ oxygen rate/ speed 정보를 push_back 받는 벡터 컨테이너들이다. current_env와 current_vehicle은 현재

journey의 environment와 vehicle 객체를 담고 있는 데이터멤버들이다. 마지막으로 text_graphic은 Journey constructor에서 인자로 받아온 string데이터이다(추후 text_graphic을 print할 때 사용된다.).

멤버 function들을 다음과 같다.

```
void move(int mode)
```

```
void change_env(void){
```

```
void finish_journey(char cause)
```

```
void initial_status_print(void){
```

```
bool end_condition(void){
```

```
void print_black_box(void){
```

```
void print_final_status(void){
```

move는 main함수에서 계속해서 사용해주는 멤버 function으로, 인자인 int mode를 사용해 current_vehicle의 move_1() 혹은 move_2() function을 invoke한다. 또한, 여기서 move 이전/ 이후의 vehicle 상태에 따라 다른 함수들(ex. change_env, finish_journey 등)을 이용해 추가적인 처리작업을 해준다(여기서, 처리작업들을 모두 stack의 push와 pop형태로 이루어짐에 유의해야한다 - 순차적으로 control이 넘어가는 것으로 착각해 오랫동안 고생하며 디버깅했던 포인트).

change_env는 현 Journey class의 Environment vector에 대해 루프를 돌아주는 멤버 function인데, 함수 이름에서 알 수 있듯, 다음 환경으로 상태변화를 시켜주는 기능을 한다. 여기서 유의할 점은, unexpected environment인 X와 Y의 경우, 다른 environment케이스에서와는 다르게, static_cast 사용 및 새로운 vehicle 객체를 생성하여 current_env와 current_vehicle값을 변화시키지 않고, 과제스펙에서 명시한 내용들을 직접 수행한다는 것이다 (X와 Y 외 다른 환경들은 모두 change_env가 아닌 move 함수에서 vehicle 내 정의된 멤버함수들이 중점적으로 문제를 처리해준다.).

```
else if (env_type=="X") {
    global_unexpected = true;
    if (extra_mode_num==2){
        vector_count++;
        if ((unsigned)vector_count == env_list.size()) last_loop = true;
        return;
    }
    srand((unsigned int)time(NULL));
    int random_num = rand()%100; // 0~99사이의 수 생성
    if (random_num<20){ // 0~19 이면 (20/100 -> 20%) vehicle stop
        current_vehicle->stop_vehicle();
    }
    else{
        total_energy-=100;
        if (total_energy<=0) total_energy=0;
    }
}
```

(참고) unexpected environment들의 경우, 특정 퍼센트의 가능성으로 어떠한 일이 발생한다. 이러한 확률성 상황처리를 위해 난수생성을 해주어야 했는데, 짧은 주기에 따라 생성한 random number가 반복되는 일을 방지하기 위해 매번 현재 시간을 seed값으로 전달해주었다.

finish_journey는 journey가 끝날 때 과제스펙에서 명시한 내용을 출력하기 위한 멤버 function이고, initial_status_print()는 처음에 Test Case가 선택되었을 때 move mode number를 입력받기 전 가장 초기 상태를 출력하기 위한 함수이다. end_condition은 현재 Journey가 끝날 상황적 요건인지를 boolean 값으로 반환해주고, print_black_box는 Journey가 끝날 때 맨 마지막에 블랙박스를 출력해주는 역할을 한다. print_final_status()는 맨 마지막 status를 출력하기 위해 따로 정의해준 함수인데, private member function으로, finish_journey함수 내부 맨 첫줄에 위치하고 있다(누락의 위험성을 줄이기 위함).

⑤ class 구성 (2) - Vehicle Class (+ Car, Airplane, Submarine)

class 구성 (2)에선 Vehicle class가 base class이며, Car, Airplane, Submarine은 해당 base class를 상속받는 derived class이다.

Vehicle class의 구조를 먼저 살펴보면 다음과 같다.

```
// parent class
class Vehicle {

    public:
    Vehicle(string graphic,int& index, int speed, int temperature, int& oxygen, int& energy_value, int local_distance, int& total_dist)
    : graphic_index(index), oxygen_rate(oxygen), energy(energy_value), total_distance(total_dist){
        this -> graphic = graphic;
        this -> speed = speed;
        this -> temperature = temperature;
        this -> local_distance = local_distance;
        vehicle_stop = false;
        arrived = (local_distance<=0);
        fixed_local_distance = local_distance;
    }
```

```
private:
int speed;
int& energy;
int temperature;
int& oxygen_rate;
int local_distance;
int& total_distance;
bool arrived;
bool vehicle_stop;
int& graphic_index;
string graphic;
int fixed_local_distance;
```

데이터 멤버들을 보면,

energy, oxygen_rate, total_distance, graphic_index는 Journey 전반에 걸쳐 매 vehicle들이 매 environment마다 공유해야 하는 (지속적으로 값이 이어져야 하는) 데이터이므로 reference형을 이용해 값을 저장하고, 조작할 수 있도록 한다. fixed_local_distance는 trouble shooting process에서 선언 이유에 대해 더 자세히 설명되어있으나, 먼저 간단히 설명하면 생성자에서 처음 만들어졌을 때 받아온 local distance의 초기값이다. 이 값은 vehicle이 생성되고 나서 어떠한 조작을 가해주더라도 절대 값이 변하지 않는다. 따라서, 조건문 등에서 보다 안정적으로 사용할 수 있다.

멤버 function들을 보면(기본 get/set function들에 대한 설명은 생략한다), 다음과 같은데,

```
virtual void update_all_losses(void)=0;
virtual void move_1(void)=0;
virtual void move_2(void)=0;
virtual void print_status(void)=0;


bool end_condition(void){
    if (vehicle_stop) return true;
    if (local_distance<=0) return true;
    if (energy<=0) return true;
    if (oxygen_rate<=0) return true;
    return false;
}
char end_cause(void){
    if (vehicle_stop) return 'S';
    if (local_distance<=0) return 'A';
    if (energy<=0) return 'E';
    if (oxygen_rate<=0) return 'O';
    else return 'X';
}
void stop_vehicle(void){ vehicle_stop = true; } // for unexpected cases
void increment_graphic_position(void){graphic_index++;}
void print_graphic(void){cout<<graphic.substr(0, graphic_index) + "@" + graphic.substr(graphic_index)<<endl;}
```

먼저, 상단에서 볼 수 있듯 총 4개의 pure virtual function을 지니고 있다. 이는 각 vehicle의 derived class들에서 과제스펙에 명시된 적절한 내용대로 over-riding 해줄 함수들이다. 그 외에 end_condition은 현재 vehicle이 멈출 상황인지를 boolean 값으로 반환해주고, end_cause는 end_condition이 충족될 경우 끝나게 된 원인이 무엇인지를 char 이니셜 값으로 반환해준다. 그 아래 stop_vehicle()함수는 unexpected environment를 만났을 경우 확률에 따라 외부에서 vehicle을 stop시켜 주기 위해 정의해주었다. increment_graphic_position은 한 unit씩 앞으로 나아갈 때마다 text_graphic 상의 vehicle 위치를 업데이트시켜주기 위해, print_graphic은 constructor 인자에서 받아온 graphic_index reference값을 이용하여 현재 vehicle 위치를 포함한 text_graphic을 출력하기 위해 설정해준 함수이다.

다음으로 Vehicle class의 derived class들(Car, Airplane, Submarine)은 각각의 차이점들을 중심으로 살펴보도록 하자.
Car class는

```
void solar_panel_recharge(void){
    if (humidity < 50){
        if (get_energy() + 200 > 1000) set_energy(1000); // Energy cannot be over 1000
        else set_energy(get_energy() + 200);
    }
}
```

Vehicle derived class들 중 유일하게 energy를 충전할 수 있는 solar panel recharge 함수를 지니고 있다. 본 함수는 vehicle이 Road_Environment에 진입했을 때 딱 한번 실행되며, 만약 Journey class에서 solar panel 충전기가 고장 났다

면, 그 이후부터는 본 함수가 호출되지 않도록 Journey class 차원에서 경우처리를 해주었다.

```
virtual void update_all_losses(void){update_energy_loss();}
```

매 unit마다 업데이트 시켜줘야 할 loss 정보는 energy 관련 loss밖에 없다. 어떠한 경우에 어느 만큼의 energy가 감소되는지에 대한 정보는 과제스펙을 그대로 따랐다.

Airplane class는

```
void update_speed_loss(void){
    if ((air_density >= 30)&&(air_density < 50)) set_speed(get_speed()-200);
    else if ((air_density >= 50)&&(air_density < 70)) set_speed(get_speed()-300);
    else if (air_density >= 70) set_speed(get_speed()-500);
}

void update_oxygen_rate_loss(void){
    set_oxygen_rate(get_oxygen_rate()-((altitude/1000)*10));
    if (get_oxygen_rate() <= 0) set_oxygen_rate(0);  // oxygen cannot go below 0
}
```

car에서와는 달리 speed loss와 oxygen rate loss 관련 함수를 지니고 있다. update_speed_loss()는 Airplane class constructor가 호출될 때 딱 한번 실행되고, update_oxygen_rate_loss()는 update_energy_loss()와 더불어 매 unit단위로 움직일 때마다 호출된다.

```
virtual void update_all_losses(void){
    update_energy_loss();
    update_oxygen_rate_loss();
}
```

따라서 Airplane class의 loss update는 energy와 oxygen rate, 두 property 모두에서 이루어져야 한다.

Submarine class는

```
void light(void){
    set_energy(get_energy()-30);
    if (get_energy()<=0) {set_energy(0);}  // energy cannot go below 0
}
```

타 derived class들에서와는 달리 light function이 있는데, 이는 매 unit 이동마다 발동되고, energy loss와 관련된 함수이므로

```
void update_energy_loss(void){
    light();

    if ((0 < get_temperature())&&(get_temperature()< 40)){set_energy(get_energy()-5);}
    else if (get_temperature() >= 40) {set_energy(get_energy()-10);}
    else if (get_temperature()==0) {set_energy(get_energy()-8);}

    if (get_energy()<=0) {set_energy(0);} // energy cannot go below 0
}
```

혹여나 누락시키는 경우가 없도록 update_energy_loss 함수 안에 위치시켜준다.

세 derived class의 move_1, move_2 함수의 구조는 구체적인 unit값을 제외하고는 동일한데, Car class 내 move_1, move_2 함수 코드를 통해 작동원리를 간단히 살펴보도록 하자.

```
virtual void move_1(void){ // per unit
    set_local_distance(get_local_distance()-50);
    set_total_distance(get_total_distance()+50);
    update_all_losses();
    increment_graphic_position();
    if (!global_unexpected) cout<<"Successfully moved to next "<< 50 <<" km"<<endl;
}

virtual void move_2(void){
    int before_move = get_total_distance();
    for (int i=0; i < ((get_fixed_local_distance()/50)+1); i++){
        if (!end_condition()){
            set_local_distance(get_local_distance()-50);
            set_total_distance(get_total_distance()+50);
            update_all_losses();
            increment_graphic_position();
        }
    }
    if (!global_unexpected) cout<<"Successfully moved to next "<< get_total_distance()-before_move  <<" km"<<endl;
}
```

move_1의 경우, 한 unit 만큼만 움직이므로 Car의 unit값인 50을 local distance (여기서 local distance는 사실상 left-over local distance를 의미한다. 따라서 vehicle이 움직임에 따라 local_distance값은 자연스레 감소된다.)로부터 빼준다. 또한 total_distance에는 unit값(50)만큼을 더해준다. 한 유닛만큼 이동할 때마다 loss정보들을 업데이트 시켜주며, graphic 상의 position또한 1만큼 증가시켜준다. 그리고 만약 현재 거쳐 가고 있는 환경이 Environment_X, Environment_Y가 아니라는 조건 하에 Successfully moved to next (...) km 문장을 출력해준다.

move_2의 경우, 아직 이동을 수행하기 전 total_distance의 값을 before_move라는 int 변수에 저장해준다. 이는 추후 Successfully moved to next (...) km 문장 출력시 (...)에 해당하는 적절한 값을 계산해 표시해주기 위한 작업이다. 다음으로는 fixed local distance를 unit으로 나눠준 횟수만큼의 for loop를 돌아주는데, for

loop 내부를 보면 move_1 함수의 코드와 거의 동일한 것을 확인할 수 있다. 하지만, 중간에 end_condition이 만족되면 루프 내 실행될 코드가 없어지며, end_condition이 만족되기 직전까지의 거리로만 나아간다. 마지막 출력문장의 구조는 move_1에서와 동일하다.

⑥ class 구성 (3) - Environment Class
(+ Road_Environment, Sky_Environment, Ocean_Environment, Environment_X, Environment_Y)

class 구성 (3)에선 Environment class가 base class이며, Road_Environment, Sky_Environment, Ocean_Environment, Environment_X, Environment_Y은 해당 base class를 상속받는 derived class이다.

```
class Environment{
    private:
    string type_name;
    int temperature;
    int distance;

    public:
    Environment(int dist, int temp, string name="undefined"){
        distance = dist;
        temperature = temp;
        type_name = name;
    }
    int get_temperature(void){return temperature;}
    int get_distance(void){return distance;}
    string get_type_name(void){return type_name;}
};
```

Environment base class와 derived class들은 테스트 케이스 인풋 정보를 추출하고, 환경 정보들을 Journey class에 전달해준 후, 보유하고 있는 데이터들을 통해 각 환경마다의 vehicle들을 적절히 instantiation시켜주기 위한 기능 외엔 특별한 역할을 하지 않기 때문에, 코드가 굉장히 간결하다.

하나 유의할 점은 string type_name 프로퍼티인데, Journey class에서 각 환경의 종류마다 각기 다른 처리를 해줘야 하기에, 각 Environment 이름을 담고 있을 string형 type_name을 정의해주었다(base class environment 이름은 undefined로 초기화 해 주었다).

```
class Environment_X: public Environment{
    private:
    int temperature;
    int distance;

    public:
    Environment_X(int dist, int temp):Environment(dist, temp, "X"){}
};

class Environment_Y: public Environment{
    private:
    int temperature;
    int distance;

    public:
    Environment_Y(int dist, int temp):Environment(dist, temp, "Y"){}
};
```

특히, unexpected environment X,Y는 해당 Environment_X, Environment_Y class에서 본격적인 처리작업을 진행하지 않을 것이기에, Environment base class를 상속받기 위한 최소한의 코드만 작성해주었다(따라서, input에서 정보를 추출해 Environment 객체 포인터들로 이루어진 env_list 벡터를 반환할 때도, Environment X와 Y엔 0,0이라는 의미 없는 dummy data가 들어갔다. 이는 X, Y 또한 env_list에 넣기 위해 불가피하게 취해야만 했던 방식이다).

이 외 Road, Sky, Ocean environment는 각자만이 지니는 고유한 값(ex, humidity, altitude, depth 등)이 constructor 코드와 data member 정의부문에 추가된다 + 그에 대한 get function을 선언해준다는 사항 외엔 세 클래스 정의 내용이 base class 코드와 거의 동일하기에, 보다 자세한 설명은 생략하도록 한다.

```
class Road_Environment: public Environment{
    private:
    int temperature;
    int humidity;
    int distance;

    public:
    Road_Environment(int dist, int temp, int humd):Environment(dist, temp, "Road"){
        humidity = humd;
    }
    int get_humidity(void){return humidity;}
};
```

(ex. 참고: Road_Environment에 특정적으로 추가해준 humidity 프로퍼티)

3. Troubleshooting points

①

```
virtual void move_2(void){
    int before_move = get_total_distance();
    for (int i=0; i < ((get_fixed_local_distance()/50)+1); i++){
        if (!end_condition()){
            set_local_distance(get_local_distance()-50);
            set_total_distance(get_total_distance()+50);
            update_all_losses();
            increment_graphic_position();
        }
    }
    if (!under_unexpected()) cout<<"Successfully moved to next "<< get_total_distance()-before_move  <<" km"<<endl;
}
```

두 번째 move 모드에서, status change 훨씬 전에, 중간쯤까지만 가다가 멈춰버리는 문제가 발생했었다. 이는 get_fixed_local_distance()대신 그냥 get_distance()/50을 for loop의 조건으로 사용했을 때 발생한 문제인데, 계속해서 local_distance를 줄여나가기 때문에 조건에서 사용되는 값이 고정되지 않고, 점점 작아져 기존에 수행해야 하는 반복횟수를 만족시키지 못하는 결과를 초래했다. 이러한 문제를 해결해주기 위해, 생성자가 불려지는 시점의 local_distance값을 fixed_local_distance라는 int형 private멤버에 저장해두고, 추후 조건문에서 사용할 때 get function을 이용해 해당 값을 활용할 수 있도록 하였다.

②

```
virtual void print_status(void){
    cout<<"Current Status: "<<"Car"<< endl;
    cout<<"Distance: "<< get_total_distance() <<" km"<<endl;
    cout<<"Speed: "<< get_speed() <<" km/hr"<<endl;
    cout<<"Energy: "<<  get_energy() << endl;
    cout<<"Temeperature: "<< get_temperature() <<" C"<<endl;
    cout<<"Humidity: "<< humidity <<endl;
}
```

base class 생성자를 통해 초기화 해준 변수들을 get function을 통하지 않고 직접적으로 써줬더니(ex. temperature, energy, speed) 엉뚱한 큰 값이 출력되는 문제가 발생했다. 이는 접근제한자의 조건을 고려하지 못하여 발생한 문제로, 해당 멤버들에 대해 정의한 get function을 호출하여 문제를 해결하였다.

```
void update_speed_loss(void){
    if (air_density >= 30) set_speed(get_speed()-200);
    else if (air_density >= 50) set_speed(get_speed()-300);
    else if (air_density >= 70) set_speed(get_speed()-500);
}
```

비슷하게, 값을 변경해주어야 할 때 get, set function을 사용하지 않고 explicitly 값을 불러와 speed-=200;와 같은 식으로 처리해주니, 원하는 값이 제대로 할당되지 않는 문제가 발생했다. 이 또한 get/set function을 활용하니 다시 값을 처음에 의도한 대로 유효하게 설정할 수 있게 되었다.

③

```
private:
int speed;
int& energy;
int temperature;
int& oxygen_rate;
int local_distance;
int& total_distance;
bool arrived;
bool vehicle_stop;
int& graphic_index;
string graphic;
bool going_through_unexpected;
int fixed_local_distance;
```

다음으로는, class 내 reference 타입의 private member를 선언해주었는데, constructor에서 일반 data member들과 같이 { } 블록문 안에 값들을 할당해주었더니 오류가 발생하였다. reference 타입은 반드시 선언과 동시에 초기화 되어야 하는데, 이를 고려하지 않았기 때문에 발생한 문제로, 다음 이미지자료에서와 같이 member initializer 형태로 처리해 문제를 해결할 수 있었다.

```
// parent class
class Vehicle {

    public:
    Vehicle(string graphic,int& index, int speed, int temperature, int& oxygen, int& energy_value, int local_distance, int& total_dist)
    : graphic_index(index), oxygen_rate(oxygen), energy(energy_value), total_distance(total_dist){
        this -> graphic = graphic;
        this -> speed = speed;
        this -> temperature = temperature;
        this -> local_distance = local_distance;
        vehicle_stop = false;
        arrived = (local_distance<=0);
        going_through_unexpected = false;
        fixed_local_distance = local_distance;
    }
```

4. Screen-shots of the interactive console

① Normal Mode

```
string TC3 = "[R1000T20H49],[Y],[S1000T20H49A1000D0],[R150T20H49]";
```

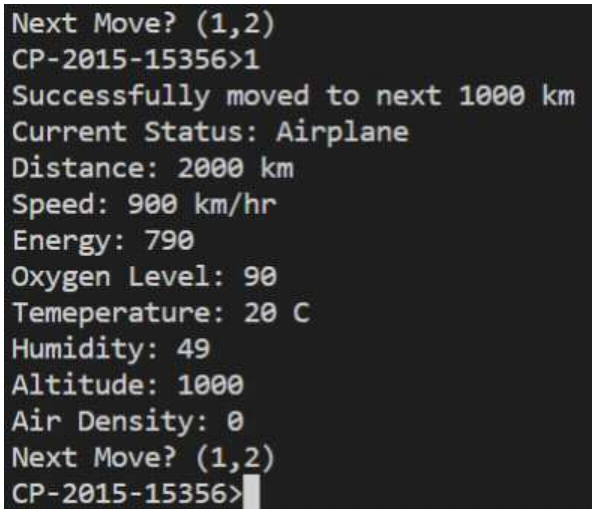
Normal 모드에서 TC3을 선택해 journey를 시작해보자.

```
Mode Select(1 for EXTRA, 2 for NORMAL) :2

PJ1.LJH.2015-15356
Choose the number of the test case (1~10) : 3

Testcase #3.

Current Status: Car
Distance: 0 km
Speed: 0 km/hr
Energy: 1000
Temeperature: 20 C
Humidity: 49
Next Move? (1,2)
CP-2015-15356>
```

프로그램을 처음 실행할 때, Mode Select 값에 2를 입력해 NORMAL mode가 activate되도록 하였다. 3번 테스트케이스를 선택했고, 선택한 Test case 번호와 더불어 초기 상태가 출력된다.

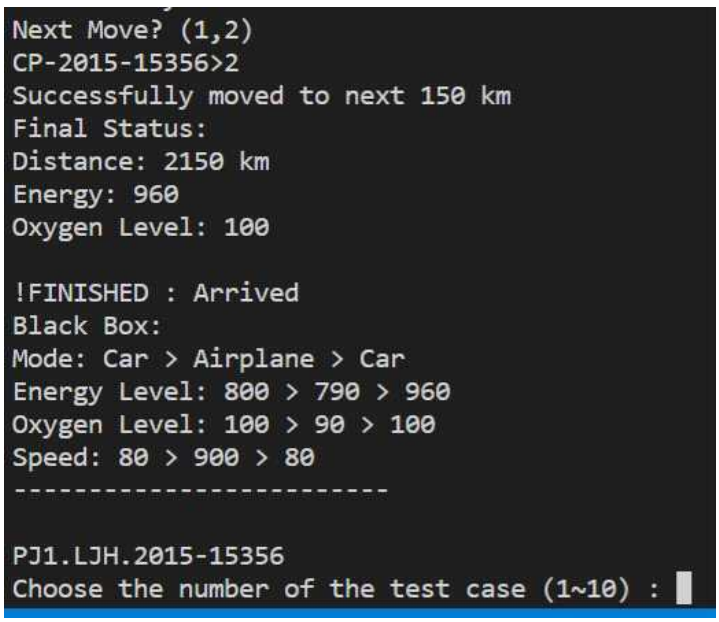
```
Next Move? (1,2)
CP-2015-15356>2
Successfully moved to next 1000 km
Current Status: Car
Distance: 1000 km
Speed: 80 km/hr
Energy: 800
Temeperature: 20 C
Humidity: 49
Next Move? (1,2)
CP-2015-15356>
```

Move mode는 2를 선택해주었다. 따라서 status change가 일어나기 전까지 이동한다. 이 환경에서 Car은 한 유닛을 움직일 때마다 energy -10 이 이루어지므로, 처음 시작 1000부터 10*20 =200을 뺀 값, 즉 800이 나오는 게 옳다.



다음으로는 한 유닛씩만 이동하는 Move mode 1을 선택해주었다. sky environment로 환경이 바뀌었기에, 한 유닛이 1000km인 Airplane으로 status가 변경되었다. (원래는 Road와 Sky 사이에 [Y] unexpected environment가 존재하지만, 현재는 Extra가 아닌 Normal 모드이기에 그냥 무시하고 바로 Sky environment로 넘어갔다.) Sky environment의 T와 H값을 고려하면, 한 유닛 이동 당 10씩의 energy가 감소되는 게 맞다. 이 바로 직전 state에서의 energy가 800이었으므로, 현재는 790이 나오는 것이 올바르다.

speed loss를 계산해보면, 현재 Air Density가 0이니 speed에는 변화가 없다. 마지막으로 oxygen_rate 감소치를 보면, 주어진 환경의 altitude가 1000이니 한 유닛 이동 당 10씩 감소해야 한다. 기존 oxygen level이 100이었는데 현재는 90이므로, 올바르게 감소했음을 확인할 수 있다.



다음 Move mode로는 2를 선택해 주었다. 다음 environment가 마지막 환경이었으므로, 2를 선택했을 때, 중간에 에너지/산소 고갈이 없으면 arrived로 Journey가 끝이 나게 된다. Journey가 끝날 때 Final Status와 Arrived 부분이 잘 출력됨을 확인할 수 있다. 한 Journey가 끝나면 0으로 프로그램을 끝내지 않는 이상, 계속해서 테스트 케이스 번호를 받는다.

```
PJ1.LJH.2015-15356
Choose the number of the test case (1~10) : 0

C:\Users\82102\AppData\Roaming\Code\User\cpp_test>
```

(test case 번호에 0을 입력하면 프로그램이 종료된다)

② Extra Mode

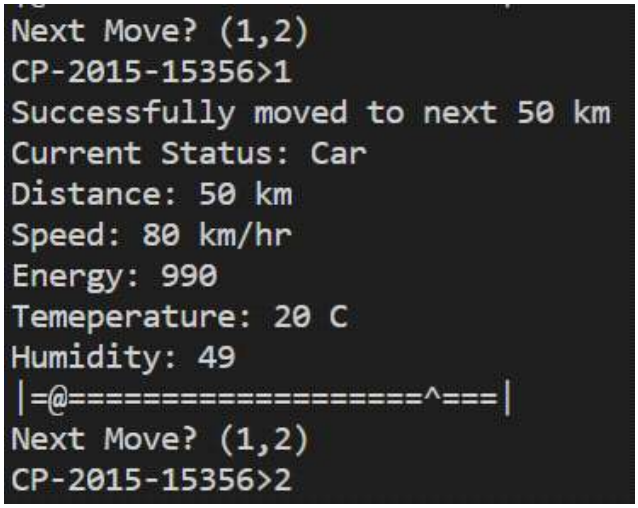
```
string TC3 = "[R1000T20H49],[Y],[S1000T20H49A1000D0],[R150T20H49]";
```

똑같은 테스트케이스를 이용해 Extra Mode를 시험해보자.

```
Mode Select(1 for EXTRA, 2 for NORMAL) :1

PJ1.LJH.2015-15356
Choose the number of the test case (1~10) : 3

Testcase #3.

Current Status: Car
Distance: 0 km
Speed: 0 km/hr
Energy: 1000
Temeperature: 20 C
Humidity: 49
|@====================^===|
Next Move? (1,2)
CP-2015-15356>1
```

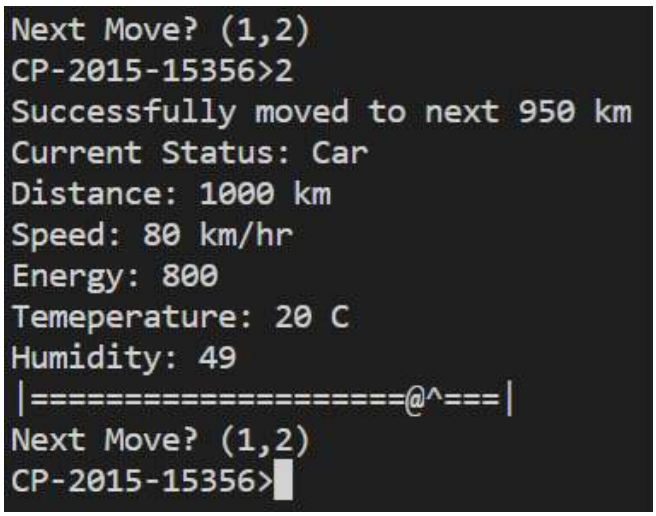
프로그램을 처음 시작할 때, 1을 입력해 Extra모드를 선택한다.
테스트 케이스 번호를 선택하고 그에 따른 initial state를 출력할 때, text_graphic이 함께 출력되는 것을 확인할 수 있다.

첫 이동 모드는 한 유닛만 움직이는 1 모드를 선택했다.



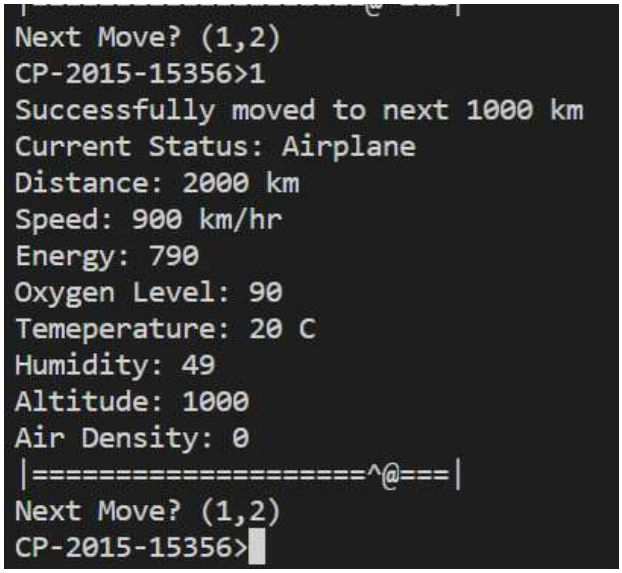
text graphic에서의 골뱅이(vehicle) 위치가 한 칸 앞으로 이동한 것을 확인할 수 있다. 또한 한 유닛 당 움직일 때 현재 환경에서는 energy가 10씩 감소해야하므로, 총 990이 출력되는 것이 맞다.

이다음으로는 2번 모드로 move를 수행해본다.



현재 Road Environment의 left-over local distance만큼 모두 이동하였다. 골뱅이 위치가 첫 번째 road환경 기호 맨 뒤에 위치하고 있음을 확인할 수 있다. 또한, 총 19 유닛을 움직였으므로 에너지는 19*10= 190, 즉 990-190 = 800이 출력되는 것이 올바르다.

이 바로 뒤엔 text_graphic에서는 나타나지 않은 unexpected environment Y가 존재한다. 1을 입력해 unexpected environment Y를 어떻게 처리해주는지 확인해보자.

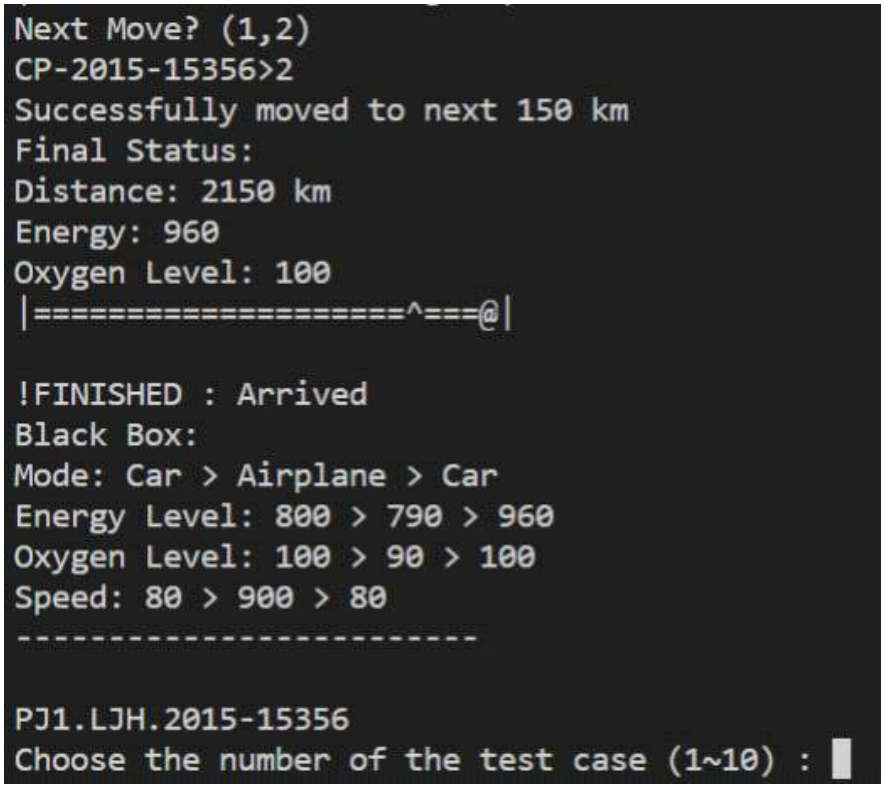


원래는 35%의 확률로 vehicle stop이 일어나 journey가 중단될 수 있으나, 나머지 65%의 확률에 걸려 vehicle stop이 일어나지 않았다.

나머지 65%의 확률에서 또 50%의 확률로 (이 바로 직전 환경이 Road이니) 앞으로의 Journey에서 존재할 모든 Car의 solar panel 에너지 충전기가 고장 나 작동하지 않을 수 있다. 혹은 아무 영향도 받지 않고 그저 다음 환경으로 넘어갈 수 있다.

solar panel 에너지 충전기 고장 여부는 맨 마지막에 다시 Road Environment를 만났을 때 energy 변화를 보고 판단할 수 있다. 아무튼 이렇게 unexpected environment를 잘 지나 정상적으로 그다음 Sky Environment에서 move 1이 수행되었다. 수치들은 모두 올바르게 나왔음을 확인할 수 있다.

현재 상태에서 Move mode로 2를 선택해 마지막 Road Environment 끝까지 가보도록 하자.



끝까지 갔고, 결국 Arrived를 이유로 Journey가 끝났다. energy change를 확인해 앞서 Solar Panel Energy Recharger가 고장 났는지 확인해보자. 주어진 스펙으로는 한 유닛당 10씩의 에너지가 감소해야 하고, 총 3단위를 움직였으니 30의 에너지가 감소한다. 여기서 solar panel recharge가 작동했다면, +200의 에너지가 충전되었겠지만, 블랙박스를 확인해보면 충전이 이루어지지 않은 채 30이 그대로 감소했음을 확인해 볼 수 있다.

Extra 모드에서 또한 한 Journey가 끝이 나면 그 다음 테스트 케이스 번호를 받는다.

(cf) 운이 나빠 unexpected situation을 거칠 때 vehicle stop이 발생하게 되면, 다음과 같이 값이 출력된다.