0. 파일

- `main.cpp` : main 함수가 포함된 파일로, 시뮬레이터를 위한 설정 상수와 시뮬레이터 구현이 main 함수 내에 있습니다.
- `simulator.h` : 시뮬레이터의 작동에 필요한 함수, 타입, 클래스 등의 선언이이 있고 최대한 한 파일 내에서 코드를 확인할 수 있도록 대부분의 정의가 작성되어 있습니다.
- `simulator.cpp` : 일부 `simulator.h`에서 바로 구현할 수 없는 함수와 메서드 정의되어 있습니다.
- `schedular.h` : 실질적으로 **수정해야 할 파일**로 Scheduler 클래스의 기본 선언이 있습니다.
- `schedular.cpp` : 실질적으로 **수정해야 할 파일**로 Scheduler 기본 선언의 구현이 작성되어 있습니다.
- `CMAKELists.txt` : CMake환경을 사용하실경우 사용하시면 됩니다. 다른 환경을 사용하실경우 지우셔도 무방합니다.

1. 해야할 것

scheduler.cpp 파일의 메서드 3개 **on_info_updated**, **on_task_reached**, **idle_action**를 완성해야 합니다.
각 메서드가 해야 할 일은 다음과 같습니다.

- **on_info_updated**
  현재 알려진 정보를 통해 스케쥴링을 진행하고 스케쥴링 결과를 **on_task_reached**와 **idle_action**에서 출력하기 위해 저장해 두어야 합니다. 제공된 파일에는 아무 작업도 하지 않도록 되어있습니다.

- **on_task_reached**
  로봇이 작업에 도착했을 때 작업을 시작할지 여부를 `bool` 타입으로 반환합니다. 제공된 코드에서는 로봇 타입이 드론이 아닐경우 무조건 작업을 실행하게 되어있습니다.

  - **주의**
    이 메서드가 `true`를 반환 할 경우 로봇이 작업을 완수할 수 있는지 여부를 확인하지 않고 작업이 시작됩니다. 작업을 완료할 수 없는 로봇이 작업을 시작할경우 로봇이 에너지를 다 쓸 때 까지 다른 로봇이 해당 작업을 시작하지 못하며 진행된 작업 내역은 초기화 됩니다.

- **idle_action**
  현재 로봇이 이동이나 작업을 진행하고 있지 않을 때 어떠한 행동을 할 지를 `ROBOT::TYPE` 타입으로 반환하는 메서드 입니다.
  총 5가지 종류의 행동 `UP`, `DOWN`, `LEFT`, `RIGHT`, `HOLD` 이 있습니다.
  지도 밖 혹은 벽으로 이동하는 행동을 반환할 경우 메시지와 함께 로봇은 제자리를 유지하게 됩니다.

파일에서 언급했듯이 schedular.h과 schedular.cpp 파일만을 수정하여 스케쥴러를 완성하여 주시기 바랍니다. 추가적으로 필요한 함수나 선언 모두 이 두 파일 내에서 진행해 주시기 바랍니다. 프로그래밍 단계에서 동작의 확인을 위해 다른 파일들을 수정하시는 것은 상관 없지만 평가시 두개의 스케쥴러 파일과 처음 제공된 나머지 파일을 통해 평가를 진행할 것이기 때문에 **반드시 처음 제공된 파일과 함께 정상 작동하여야 합니다.**

2. 주어지는 정보

- `const ROBOT &robot` : 로봇 객체의 레퍼런스입니다. 사용 가능한 정보는 다음과 같습니다.

```
robot.id // int형인 로봇의 id
robot.type // ROBOT::TYPE형인 로봇의 타입
robot.get_coord() // Coord형인 로봇의 현재 위치
robot.get_status() // ROBOT::STATUS형의 로봇의 현재 상태
robot.get_target_coord() // 로봇의 상태가 MOVING일 때 목표 좌표
robot.get_energy() // 현재 남은 로봇의 에너지
```

- `const TASK &task` : 작업 객체의 레퍼런스입니다. 사용 가능한 정보는 다음과 같습니다.

```
task.id // 작업의 id
task.coord // 작업의 위치
task.get_cost(type) // type에 해당하는 로봇이 작업을 수행할 때 필요한 에
task.get_assigned_robot_id() // 할당된 로봇의 id를 반환합니다. 할당도니
task.is_done() // 작업의 완료 여부
```

- `const set<Coord> &observed_coords` : 에너지가 남아있는 로봇들이 현재 관측하고 있는 영역의 좌표 집합

```
for(auto it = observed_coords.begin(); it != observed_coords.end(); ++it)
{
    cout << "x=" << it->x << " y=" << it->y << endl;
}
```

- `const set<Coord> &updated_coords` : known_object_map 내 정보가 업데이트 된 영역의 좌표 집합. 사용 방법은 observed_coords와 동일합니다.
- `const vector<shared_ptr<ROBOT>> &robots` : 모든 로봇의 벡터. shared_ptr<ROBOT>는 ROBOT*라고 생각하고 사용하시면 됩니다.

```
cout << robots[0]->id << " " << robots[0]->get_energy() << endl;
for(auto it = robots.begin(); it != robots.end(); ++it)
{
  cout << (*it)->id << " " << (*it)->get_energy() << endl;
}
```

- `const vector<shared_ptr<TASK>> &active_tasks` : 로봇에 의해 확인되었고 완료되지 않은 작업의 벡터입니다. 현재 로봇이 작업중인 작업 또한 포함합니다. 사용 방법은 `const TASK &task`와 `const vector<shared_ptr<ROBOT>> &robots`를 참고하시기 바랍니다.
- `const vector<vector<OBJECT>> &known_object_map` : 로봇에 의해 확인된 오브젝트 맵입니다. 로봇이 현재 관측중이지 않은 영역에 작업이 생성될 경우 이 지도에는 반영되지 않습니다.

  ```
  cout << known_object_map[x][y] << endl;
  ```

- `const vector<vector<vector<int>>> &known_cost_map` : 로봇에 의해 확인된 이동 비용 맵입니다. x,y좌표와 type을 통해 이동 비용을 확인할 수 있습니다. 확인 되지 않은 영역의 이동 비용은 -1이고 벽의 이동비용은 int의 최대값입니다.

  ```
  cout << known_cost_map[x][y][type]
  ```

  모든 정보가 모든 과정에서 필요한 것은 아니지만 편의성을 위해 필요 이상의 정보를 제공하니 적절히 필요한 정보만 사용하시기 바랍니다.

3. 타입 설명

- `OBJECT` : 지도의 특정 좌표에 있는 오브젝트를 나타내는 `enum clas`로 `EMPTY`, `ROBOT`, `TASK`, `ROBOT_AND_TASK`, `WALL`, `UNKNOWN`가 있고 로봇을 통해 확인되지 않은 영역은 `UNKNOWN`으로 표시됩니다.
- `ROBOT::TYPE` : 로봇의 종류를 나타내는 `enum class`로 `DRONE`, `CATERPILLAR`, `WHEEL`가 있습니다.
- `ROBOT::STATUS` : 로봇의 상태를 나타내는 `enum class`로 `IDLE`, `WORKING`, `MOVING`, `EXHAUSTED`가 있습니다.
- `ROBOT::ACTION` : 로봇의 행동을 나타내는 `enum class`로 `UP`, `DOWN`, `LEFT`, `RIGHT`, `HOLD`가 있습니다.

4. 정보 프린팅

다음은 디버깅과 실행 상태 출력 등을 위한 방법입니다.

- `simulator.h` 의 `#define VERBOSE`의 라인코멘트를 해제하면 로봇의 이동시작/좌표변경/이동 종료/작업시작/작업종료, 작업의 새로운생성/작업발견 그리고 매 시간 현재시간과 전체 오브젝트 맵 등의 정보가 표시됩니다.

- `main.cpp`의 `main` 함수 내에 `map`인스턴스를 통해 다양한 지도정보를 출력 가능합니다.
    - `map.print_cost_map(type)` : type에 따른 cost map을 출력합니다.
    - `map.print_object_map()` : 전체 object map을 출력합니다.
        - 벽 : `WAL`
        - 로봇 한대 : `R{type}{id}` (예시 id 3인 드론일 경우 `RD3`)
        - 로봇 두대 이상 : `RS{number of robot}` (예시 로봇 3대가 겹쳐있을 경우 `RS3`)
        - 작업 : `T{task id}` (예시 id 3인 작업 `T03`)
        - 작업과 로봇 한대 : `T{robot type}{robot id}` (예시 작업과 id 2의 드론 `TD2`)
        - 작업과 두대 이상의 로봇 : `TS{number of robot}` (예시 작업과 로봇 3대가 겹쳐있을 경우 `TS3`)
        - 빈곳 : 빈칸
    - `map.print_known_object_map` : 로봇에 의해 발견된 known object map을 출력합니다. 최신 정보가 아닐 수 있습니다. 발견되지 않은 곳은 `UNK`로 표시됩니다.
    - `map.print_robot_summary()` : 전체 로봇들의 id, 위치, 에너지, 상태, 목표 좌표, 할당된 작업 id 등을 출력합니다.
    - `map.print_task_summary()` : 최대 생성될 수 있는 작업 수, 현재 생성된 작업수, 발견되고 완료되지 않은 작업 수, 완료된 작업 수 를 출력하고 현재 생성된 작업들에 대해 id, 위치, 할당된 로봇 id, 작업에 필요한 코스트 등을 출력합니다.

4. 알아둘점

4.1. 로봇

시야범위

- 드론 : 자신을 기준으로 위,아래,좌,우로 거리 2까지 직사각형 영역을 확인하여 총 25칸을 확인합니다.
- 캐터필러 : 자신을 기준으로 위,아래,좌,우로 거리 1까지 직사각형 영역을 확인하여 총 9칸을 확인 합니다.
- 휠 : 자신을 기준으로 위,아래,좌,우로 거리 1까지 십자가 영역을 확인하여 총 5칸을 확인 합니다.
- 에너지를 다 소모한 로봇은 관측도 불가능해 집니다.

이동

- 드론의 이동비용은 시드에 따라 달라지며 맵 전체가 동일합니다.
- 캐터필러는 최소 이동비용은 크지만 최대 이동 비용이 작습니다.
- 휠은 최소 이동비용은 작지만 최대 이동비용이 큽니다.

- 이동에 필요한 시간은 총 이동 비용을 10으로 나눈 값을 올림한 것과 같습니다. 총 이동비용은 (현재위치 코스트 + 목표 위치 코스트) / 2
- 로봇이 이동 시 현재 위치를 벗어나는데 현재 위치 코스트의 절반의 에너지를 사용하고 목표위치의 중앙에 도착하는데 목표위치 코스트의 절반을 사용하며 목표 위치 중앙에 도착하면 이동이 종료됩니다.
- 이동중에도 관측이 가능합니다.
- 로봇은 겹쳐질 수 있습니다.
- 이동중 혹은 작업중에 에너지를 모두 소모하면 그 자리에서 정지합니다.
- 이동중 혹은 작업중에는 조작이 불가능 합니다.

기타

- 드론으로 작업을 시작하면 드론은 에너지가 모두 사용될 때 까지 작업을 진행하게 되므로 주의하시기 바랍니다.

4.2 작업

- 완성되지 않은 작업, 로봇, 벽이 있는 곳에는 새로운 작업이 생성되지 않습니다.
- 작업이 완료된 장소에는 작업이 다시 생성될 수 있습니다.
- 작업을 완료하기 위해 필요한 에너지는 로봇의 종류에 따라 다릅니다.

4.3 기타

- 완성해야 하는 세 메서드의 인수로 주어진 정보 외의 정보는 모르는 것으로 간주해야 합니다. (전체 시간, 최대 작업수 등 모르는 것으로 간주)
- 맵 크기 작게해서 먼저 알고리즘을 개발하고 추후 맵 크기를 키워 시도해 보시는걸 추천드립니다.
- 코드는 윈도우의 cl컴파일러를 통해 C++14 버전에서 작성 및 테스트 되었습니다. 다른 환경에서 문제가 발생할 경우 제보해 주시기 바랍니다.