Zadanie J - Kosmiczna ucieczka

Potężna rasa starożytnych maszyn - Żniwiarzy (ang. Reaper) - atakuje naszą Galaktykę!

Dowodzisz niewielkim statkiem kosmicznym i Twoim zadaniem jest dotarcie do głównej floty obrońców Galaktyki która stacjonuje w jednym z licznych systemów gwiezdnych. Podróż odbywa się za pomocą sieci przekaźników które wystrzeliwują statki z ogromną, nadświetlną prędkością do innego, pobliskiego systemu gwiezdnego. Musisz jednak wybrać taką trasę by nie spotkać się z flotą wroga - która, podobnie jak Ty porusza się przeskakując z układu do układu za pomocą tej samej sieci przekaźnków.

Ponadto, Żniwiarze pozostawiają za sobą okręty patrolujące w każdym systemie który odwiedzili. Jeśli spotkasz się nawet tylko z tym patrolem czeka Cię dość paskudny koniec...

Twój statek jest już wyposażony w odpowiednie algorytmy i potrafi wytyczyć bezpieczną trasę. Ponieważ jednak wróg korzysta ze skomplikowanych systemów infiltracji komputerowej, istneje niebezpieczeństwo że zaproponowana trasa jest niepoprawna i może zaprowadzić wprost w szpony Żniwiarzy. By uniknąć takiego scenariusza, postanowiłaś/eś napisać własny program weryfikacyjny i uruchomić go na niezależnym laptopie, całkowicie odłoczonym od sieci statku.

Ponieważ Twoje oprogramowanie może się przydać w przyszłości do innych zadań, musi być ono dobrze zorganizowane, podzielone na obiekty i klasy.

Na etapie projektowym powstały następujące klasy:

```
class Galaxy {
  public:

    Galaxy& load(std::istream& in);
    Galaxy& add(Reaper* r);
    Galaxy& add(Hero* h);
    Result simulate();

    class Starsystem {
      public:
      Starsystem& load(Galaxy* g, std::istream& in)
      Starsystem* adjacent(size_t idx);
      bool isTarget();
    };

    Starsystem* getSystem(size_t idx);
    ~Galaxy();
}
```

Klasa `Galaxy` opisuje galaktykę. Składa się ona z serii układów gwiezdnych (`Starsystem`), każdy o identyfikatorze podanym jako liczba całkowita.

- `load(in)` – wczytuje ze strumienia `in` opis galaktyki. Opis wygląda następująco:

  W pierwszej linijce znajdują się dwie liczby N i T. N to liczba układów gwiezdnych, T to indeks układu docelowego, gdzie stacjonuje flota obrońców. Układy numerowane są od 0.

  W kolejnych N liniach znajduje się opis każdego z układów. Linia i opisuje układ o indeksie i. Najpierw pojawia się liczba E – liczba dostępnych połączeń. Następne E liczb to numery docelowych układów do których można podróżować. Każde połączenie jest jednokierunkowe.

Metoda ma zwrócić referencję do aktualnego obiektu (`*this`)

Wszystkie liczby oddzielone są spacjami, ewentualnie znakiem końca linii.

- `add(r)`, `add(h)` – dodaje odpowiednio informację o Żniwiarzu i o Twoim okręcie, opis których jest podany osobno (poniżej). Ty jesteś jeden, ale Żniwiarzy może być wielu. Nigdy ten sam obiekt nie będzie podany dwukrotnie. Podobnie jak `load`, te metody powinny zwrócić referencję do obiektu na którym zostały wywołane.
- `simulate()` – symuluje zachowanie Żniwiarzy i zwraca rezultat `Result` (o czym za chwilę)
- `getSystem(idx)` – zwraca wskaźnik do obiektu który opisuje układ gwiezdny o indeksie `idx`. Układy indeksowane są od 0.

Klasa wewnętrzna `Starsystem` opisuje pojedyńczy układ gwiezdny. Musi ona zawierać następujące metody:

- `load(g, in)` wczytuje ze strumienia `in` opis pojedyńczego układu, tak jak wyjaśnione jest powyżej. Argument `g` to galaktyka w której system się znajduje.
- `adjacent(idx)` zwraca sąsiedni układ gwiezdny do którego można się dostać połączeniem o indeksie `idx`. Indeksy odpowiadają pozycjom na których wymienione zostały połączenia podczas wczytywania za pomocą `load`. Połaczenia indeksowane są od 0.
- `isTarget()` zwraca `true` jeśli w danym systemie znajduje się flota obrońców.

```
class Ship {
  void setPath(std::istream& in);
  Galaxy::Starsystem* start(Galaxy* g);
  Galaxy::Starsystem* advance();
  virtual ~Ship();
}
```

Klasa `Ship` opisuje planowane zachowanie statków, zarówno Twojego jak i Żniwiarzy.

- `setPath(in)` wczytuje ze strumienia informację o planowanych ruchach statku. W przypadku Twojego, jest to trasa zaproponowana przez komputer pokładowy. W przypadku Żniwiarzy jest to przewidywana trasa, która nie ulegnie zmianie w wyniku Twojej obecności. Żniwiarze zajęci są niszczeniem całych cywilizacji i nie są specjalnie zainteresowani by gonić jeden mały okręt – no chyba, że ten wleci wprost w ich szpony.

  Opis trasy podany jest w postaci liczby K – liczby wpisów. Kolejna liczba to numer systemu początkowego. Następujące po nim *K-1* liczby to kolejne numery połączeń które będą kolejno wzięte.
- `start(g)` umieszcza statek w początkowym systemie i zwraca ten system przez wskaźnik do `Starsystem`.
- `advance()` przemieszcza statek do kolejnego systemu na swej trasie i zwraca nową pozycję.

```
enum class Result {
  Invalid,
  Success,
  Failure
};

void print_result(Result r);
```

Wartość typu `Result` opisuje wynik symulacji. Możliwe wartości to:

- `Invalid` – dokonano ruchu niedozwolonego, np. w systemie S wzięto połączenie o numerze n, podczas gdy S nie ma tyle połączeń. Połączenia indeksowane są od 0. Błędem jest też, jeśli statek skutecznie przebył całą zaplanowaną trasę, nie spotkał Żniwiarzy, ale system docelowy nigdy nie został osiągnięty.
- `Success` – system docelowy został w pewnym momencie osiągnięty, a na całej poprzedzającej go tracie nie napotkano Żniwiarzy ani ich patroli. Sukcesem oznaczamy też trasę jeśli Żniwiarzy (lub ich patrol) spotkamy dokładnie w systemie docelowym. Nie interesują nas ewentualne spotkania które mogłyby nastąpić później (planowana trasa nie musi kończyć się w systemie docelowym, ale może być dłuższa). Podobnie, nie interesują nas ewentualne błędy w trasie następującej po osiągnięciu systemu docelowego.
- `Failure` – napotkano Żniwiarzy (lub ich patrol) zanim osiągnięto system docelowy.

Funkcja `print_result` wypisuje odpowiednio `Invalid`, `Success` lub `Failure` (zakończone znakiem nowej linii) na standardowe wyjscie.

Symulacja

Początkowo statki nie znajdują się nigdzie. Podczas symulacji, w pierwszym kroku umieszczane są one na pozycjach początkowych. W kolejnych krokach wszystkie statki poruszają się na kolejną pozycję. W każdym kroku należy sprawdzić warunki zadania, czy doszło do spotkania Twojego statku ze Żniwiarzami lub ich patrolami, oraz czy został osiągnięty układ końcowy.

Trzeba też sprawdzić błędy:

- Kroki Żniwiarzy są poprawne, ale długość ścieżki może być krótsza niż ścieżka jaka jest potrzebna do osiągnięcia celu. Gdy opis się kończy, można przyjąć że statek już nigdzie się nie porusza.
- Kroki zaproponowane dla Twojego statku mogą być niepoprawne: mogą pojawić się użycia połączeń które nie istnieją. Może też ścieżka skończyć się zanim osiągnięty zostanie cel. W obu tych przypadkach poprawnym wynikiem funkcji `Galaxy::simulate()` jest `Result::Invalid`. Natomiast można założyć, że wejście które służy do wczytania ścieżki jest poprawne pod względem syntaktycznym: jest odpowiednia liczba liczb, nie ma dodatkowych napisów. Liczby są nieujemne i nie są na tyle duże, żeby nie mogły się zmieścić do typów `int` lub `size_t`.

Przykład

Opis Galaktyki:

```
6 5
2 1 4
2 3 2
1 0
1 2
2 3 5
1 2
```

Opis Żniwiarzy (1 statek):

```
2 3 0
```

- Ścieżka składa się z 2 systemów
- *3* to numer początkowego systemu
- *0* to numer połączenia wziętego z systemu *3*, który – na postawie opisu Galaktyki – prowadzi do układu o numerze *2*.

Opis trasy przygotowanej dla Twojego statku:

```
4 0 1 1 0
```

- Ścieżka składa się z 4 systemów
- *0* to numer początkowego systemu
- Kolejne systemy to: *4*, *5*, *2*.

W systemie *2* spotkasz się ze Żniwiarzami, ale wcześniej zostanie osiągnięty układ docelowy (*5*). Tak więc poprawna wartość zwrócona przez `Galaxy::simulate()` to `Result::Success`.

Sprawy Techniczne

Wraz z niniejszym zadaniem dostępny jest szkielet projektu: po10.zip, Szkielet zawiera:

- Katalog `src` zawierające pliki źródłowe szkieletu opisanego w tym zadaniu
- `Makefile` który automatycznie znajdzie wszystkie pliki `.cpp` występujące w katalogu źródłowym i je skompiluje. Wygenerowany program zostanie zapisany pod nazwą `bin/po10`.
- Katalog `MSVC` zawierający projekt dla Visual Studio – jeśli wolicie takie środowisko.
- Katalog `src` zawiera też przykładowy plik `main` który wczyta test ze standardowego wejścia wywołując metody klasy `Galaxy` i `Ship`. Jeśli obsługa strumienia `std::istream` jest dla Was obca, można tam wyczytać jak przykładowo można go używać.
- Plik `0.in` który zawiera przykładowy test, który może zostać wczytany przez dostarczonego przez nas maina.

Podczas testowania mogą ulec zmianie:

- Plik z danymi

- Plik `src/main.cpp`. Będziemy testować nie tylko poprawność dla testów, ale też czy klasy mogą być wykorzystywane w inny sposób, zgodnie z wyżej wymienioną specyfikacją. Brak/Zmiana tego pliku nie powinna wpływać na działanie klas.

Wasze rozwiązanie:

- Może dodawać nowe pola i metody, zarówno publiczne jak i prywatne. Powyższy szkielet nie specyfikuje całkowidzie działań klas. Na przykład, obiekt klasy `Galaxy` wczyta informacje który system jest docelowy, ale to obiekt `Galaxy::Starsystem` zwraca informację czy jest on docelowy czy nie. Pozostaje to Waszym zadaniem by zaprojektować jak ta informacja będzie przekazana z jednego obiektu do drugiego.
- Może dodawać nowe pliki, zarówno nagłówkowe jak i `.cpp`. Testować będziemy wywołując `Makefile` który wszystkie takie pliki powinien odpowiednio uwzględnić podczas kompilacji.
- Musi zachować interfejs/specyfikację określoną w treści tego zadania i dostarczoną w szkielecie. Będziemy testować czy odpowiednie metody da się wywołać.
- Gdzieś w rozwiązaniu muszą pojawić się definicje klasy `Hero` i `Reaper`. Jeśli wykorzystacie dziedziczenie po `Ship`, ich zawartość będzie bardzo krótka. Niemniej, rozwiązania które nie korzystają z dziedziczenia też są akceptowane.
- Wszystkie pliki nagłówkowe muszą być zabezpieczne przed wielokrotnym użyciem (tzw. include guard), chyba że jest to działanie celowe
- Całość będzie kompilowana za pomocą `make release`, z `Makefilem` podanym w projekcie. Kompilacja następuje w standardzie C++11.

Jako rozwiązanie proszę spakować Wasz katalog `src` (i tylko ten katalog!) i wysłac systemem Satori.