Poszukiwanie skarbów

Celem zadania jest napisanie kodu do reprezentowania i symulowania ekspedycji poszukujących skarbów. Ponieważ uczestnicy ekspedycji, skarby, które znajdują, oraz sam przebieg ekspedycji mogą być bardzo różne, rozwiązanie powinno być ogólne.

Rozwiązanie powinno składać się z trzech plików: `treasure.h`, `member.h` i `treasure_hunt.h`.

Skarby – `treasure.h`

Skarby reprezentują wszystkie wartościowe rzeczy, które uczestnicy mogą znaleźć podczas ekspedycji. Nie wszystkie skarby są jednak łatwe i bezpieczne do wydobycia - niektóre mogą zawierać pułapkę.

Szablon klasy `Treasure<ValueType, IsTrapped>` powinien zależeć od dwóch parametrów: `ValueType` – reprezentującego typ wartości skarbu, oraz `IsTrapped` – będącego wartością logiczną wskazującą, czy skarb jest zabezpieczony pułapką. Nie powinno być możliwe stworzenie instancji klasy `Treasure` z typem wartości innym niż typ całkowitoliczbowy, czyli np. `int`, `int16_t`, `unsigned short int` itd.

Klasa powinna udostępniać:

- `Treasure(value)` – konstruktor tworzący skarb o podanej wartości,
- `evaluate()` – metoda zwracająca aktualną wartość skarbu,
- `getLoot()` – metoda zwracająca aktualną wartość skarbu i opróżniająca go (wartość skarbu staje się zerowa),
- `IsTrapped` – pole będące wartością logiczną, mówiące, czy skarb zawiera pułapkę.

Ponadto powinno być możliwe użycie skrótów:

- `SafeTreasure<ValueType>` – reprezentuje skarb bez pułapki o typie wartości `ValueType`,
- `TrappedTreasure<ValueType>` – reprezentuje skarb z pułapką o typie wartości `ValueType`.

Uczestnicy ekspedycji – `member.h`

Obietnica bogactwa zachęca wiele osób do wzięcia udziału w ekspedycji. Rozwiązanie powinno implementować dwa najczęstsze archetypy uczestników: poszukiwaczy przygód oraz weteranów.

Poszukiwacze przygód

Szablon klasy `Adventurer<ValueType, IsArmed>` powinien zależeć od dwóch parametrów: `ValueType` – będącego typem wartości zbieranego skarbu, oraz `IsArmed` – będącego wartością logiczną decydującą, czy poszukiwacz przygód jest uzbrojony. Utworzenie instancji klasy `Adventurer` powinno być możliwe tylko dla tych `ValueType`, które są poprawnymi typami wartości w szablonie `Treasure`.

Klasa `Adventurer` powinna udostępniać:

- `Adventurer()` – konstruktor bezparametrowy, tylko dla **nieuzbrojonego poszukiwacza przygód**;
- `Adventurer(strength)` – konstruktor tworzący poszukiwacza przygód o podanej sile, tylko dla **uzbrojonego poszukiwacza przygód**;
- `getStrength()` – metoda zwracająca siłę poszukiwacza przygód, tylko dla **uzbrojonego poszukiwacza przygód**;
- `isArmed` – pole będące wartością logiczną, mówiące, czy poszukiwacz przygód jest uzbrojony;
- `loot(&&treasure)` – powoduje przejęcie danego skarbu przez poszukiwacza przygód, zwiększa odpowiednio stan jego posiadania i opróżnia podany skarb; skarby zawierające pułapkę mogą być przejęte tylko przez uzbrojonych poszukiwaczy przygód i o niezerowej sile; przejęcie takiego skarbu zmniejsza dwukrotnie siłę poszukiwacza przygód;
- `pay()` – powoduje oddanie zebranych skarbów, czyli zwraca ich wartość i zeruje stan posiadania poszukiwacza przygód.

Ponadto powinno być możliwe użycie skrótu `Explorer<ValueType>` oznaczającego nieuzbrojonego poszukiwacza przygód o typie wartości `ValueType`.

Weterani

Szablon klasy `Veteran<ValueType, CompletedExpeditions>` powinien zależeć od dwóch parametrów: `ValueType` – będącego typem wartości zbieranego skarbu, oraz `CompletedExpeditions` – reprezentującego liczbę ukończonych ekspedycji w swojej karierze (wartość ta powinna być typu `std::size_t`). Utworzenie instancji klasy `Veteran` powinno być możliwe tylko dla tych `ValueType`, które są poprawnymi typami wartości w szablonie `Treasure` oraz tylko dla liczby ekspedycji mniejszej od 25 – na ukończenie większej liczby ekspedycji nie starczyłoby nikomu czasu.

Klasa `Veteran` powinna udostępniać:

- `Veteran()` – konstruktor bezparametrowy,
- `isArmed` – analogicznie jak u poszukiwacza przygód, ale weterani, znając niebezpieczeństwa ekspedycji, zawsze są uzbrojeni,
- `loot(&&treasure)` – analogicznie jak u poszukiwacza przygód, ale doświadczenie weteranów w rozbrajaniu pułapek chroni ich przed utratą siły,
- `pay()` – analogicznie jak u poszukiwacza przygód,
- `getStrength()` – analogicznie jak u poszukiwacza przygód.

Siła weterana zależy od liczby ukończonych przez niego ekspedycji – jeśli ukończył ich `n` to jego siła równa jest `n`-tej liczbie Fibonacciego (dla `n = 0` siła wynosi `0`, a dla `n = 1` siła wynosi `1`).

Pozostałe wymagania

Siła powinna być 32-bitową liczbą całkowitą bez znaku i być dostępna jako składowa publiczna `strength_t` każdej z powyższych klas.

Każdy uczestnik rozpoczyna ekspedycję bez żadnych skarbów.

Zdarzenia i ekspedycja – `treasure_hunt.h`

Każda ekspedycja składa się z szeregu zdarzeń. Dopuszczamy ich dwa rodzaje:

1. **Uczestnik znajduje skarb.** Wtedy uczestnik pozyskuje zawartość skarbu za pomocą swojej metody `loot()`.
2. **Spotkanie dwóch uczestników.** Rezultat spotkania zależy od tego, czy są uzbrojeni.
    - Jeśli żaden z nich nie jest uzbrojony, to rozchodzą się w swoje strony i nic się nie dzieje.
    - Jeśli obaj są uzbrojeni to dochodzi do pojedynku. Wygrywa go uczestnik o większej sile i zabiera on cały zebrany skarb przegranemu. W przypadku remisu, gdy uczestnicy mają równe siły, nic się nie dzieje.
    - Jeśli tylko jeden z nich jest uzbrojony, to ten drugi się poddaje. Uzbrojony uczestnik zabiera mu wówczas cały zebrany skarb.

Koncept strony spotkania

Należy zdefiniować koncept `EncounterSide<T>`. Spełnienie go przez typ `T` powinno oznaczać jedno z poniższych:

- Typ `T` reprezentuje poprawną instancję szablonu `Treasure`.
- Typ `T` reprezentuje poprawnego uczestnika ekspedycji. Za taki przyjmujemy typ, który spełnia wszystkie poniższe kryteria:
    - typ `T` udostępnia typ o nazwie `strength_t`,
    - typ `T` ma statyczne pole `isArmed`, którego typ jest konwertowalny do typu logicznego `bool`,
    - typ `T` definiuje metodę `pay()`, która zwraca obiekt, będący poprawnym typem wartości skarbu,
    - typ `T` definiuje metodę `loot(treasure)`, gdzie `treasure` jest obiektem typu `Treasure<V, B>` dla `V` zgodnego z typem zwracanym przez metodę `pay()` i `B` będącego dowolną wartością logiczną.

Zdarzenia

Zdarzenia powinny być reprezentowane poprzez typ `Encounter<sideA, sideB>`, zawierający parę referencji na obiekty odpowiednio typu `sideA` i `sideB`. Oba typy powinny spełniać koncept `EncounterSide`.

Ponadto należy udostępnić metodę `run(encounter)`, gdzie `encounter` jest obiektem typu `Encounter<A, B>`, która implementuje przebiegi wszystkich możliwych spotkań opisanych

wyżej. Nie powinno być możliwe przeprowadzenie przy użyciu tej metody żadnego innego rodzaju spotkania.

Ekspedycja

Należy zaimplementować funkcję `expedition(e1, e2, ...)`, która przyjmuje dowolną liczbę zdarzeń i przeprowadza je po kolei (w kolejności występowania argumentów).

Wymagania formalne

Rozwiązanie będzie kompilowane za pomocą polecenia

```
g++ -std=c++20 -Wall -Wextra -O2
```

Przykład użycia znajduje się w pliku `hunt_examples.cc`. Wszystkie klasy oraz funkcje w tym zadaniu powinny być konstruowalne oraz obliczalne w czasie kompilacji.

Podczas debugowania może przydać się opcja kompilacji `-fconcepts-diagnostics-depth=3`.

Pliki `member.h`, `treasure.h` i `treasure_hunt.h` należy umieścić w repozytorium w katalogu

```
grupaN/zadanie4/ab123456+cd123456
```

lub

```
grupaN/zadanie4/ab123456+cd123456+ef123456
```

gdzie `N` jest numerem grupy, a `ab123456`, `cd123456`, `ef123456` są identyfikatorami członków zespołu umieszczającego to rozwiązanie. Katalog z rozwiązaniem nie powinien zawierać innych plików. Nie wolno umieszczać w repozytorium plików dużych, binarnych, tymczasowych (np. `*.o`) ani innych zbędnych.