Zadanie 8 4–13 maja 2022 r.

kurs języka C++

macierze

Instytut Informatyki
Uniwersytetu Wrocławskiego Paweł Rzechonek

Prolog

Resource acquisition is initialization (pozyskiwanie zasobu poprzez inicjalizację), w skrócie *RAII*, to popularny wzorzec projektowy w języku C++. Technika ta jest realizowana za pomocą obiektów opakowujących zasoby (ang. wrapper) i łączy przyjęcie określonego zasobu z konstrukcją opakowania (inicjalizacja za pomocą konstruktora) a zwolnienie zasobu z destrukcją opakowania (automatycznie uruchamiany destruktor przed likwidacją obiektu). Ponieważ zagwarantowane jest automatyczne wywołanie destruktora, gdy zmienna opuszcza zasięg swojej deklaracji, to w konsekwencji zasób zostanie zwolniony w momencie, gdy skończy się czas życia zmiennej opakowującej zasób – stanie się tak również w przypadku zgłoszenia wyjątku (podczas zwijania stosu).

Techika RAII jest kluczową koncepcją przy pisaniu kodu odpornego na błędy. Mariusz Jaskółka na swoim blogu programistycznym pisze tak:

> Bardzo popularne dziś podejście do zarządzania pamięcią, jakim jest obowiązkowe użycie odkurzacza pamięci (ang. Garbage Collector) działającego w tle w języku takim jak C++ nie ma miejsca, ponieważ wiąże się z narzutem wydajnościowym, co z kolei kłóci się z koncepcją Zero Overhead Principle. Dodatkowo mechanizm ten dotyczy tylko jednego rodzaju zasobu – pamięci. I choć jest to zasób wykorzystywany przez programy najczęściej, to istnieją jednak inne, które również wymagają starannego ich zwalniania i mechanizmów, które to ułatwiają.

Zadanie

Zdefiniuj klasę opakowującą dla prostokątnej macierzy liczb zmiennopozycyjnych `double[][]` zgodnie ze wzorcem RAII. Macierz ma być opakowaniem do tablicy wskaźników na wektory; wektor natomiast ma być opakowaniem dla tablicy wartości liczbowych. Zarówno w wektorze jak i w macierzy tablice o zadanych rozmiarach mają być utworzone na stercie operatorem `new[]` w konstruktorze a w destruktorze usunięte ze sterty operatorem `delete[]`. Definicje obu klas umieść w przestrzeni nazw `obliczenia`.

```
class macierz {
    wektor **tab = nullptr; // macierz wskaźników na wektory
    const int w, k; // rozmiar tablicy: w wierszy i k kolumn
    //…
};
```

```
class wektor {
    double *tab = nullptr; // macierz liczb zmiennopozycyjnych
    const int k; // rozmiar tablicy: k pozycji
    //…
};
```

Klasy `macierz` i `wektor` nigdy nie powinny zmieniać swoich rozmiarów po utworzeniu. Klasy te mają implementować semantykę kopiowania i przenoszenia. Zgłaszaj wyjątki, gdy przekazane do konstruktorów rozmiary tablic nie będą liczbami dodatnimi. Użyj jednolitej inicjalizacji do wyzerowania tablicy zaraz po jej utworzeniu.

W klasie `wektor` zdefiniuj operatory indeksowania, zwracające odpowiednio wartość dla stałych wektorów i referencję do komórki w przypadku wektorów modyfikowalnych. Gdy indeks będzie miał wartość spoza dopuszczalnego zakresu zgłoś wyjątek. Nie zapomnij o zaprzyjaźnionych operatorach strumieniowych do czytania >> i pisania << wektorów. Zdefiniuj także operatory arytmetyczne dla dodawania i odejmowania wektorów, mnożenia wektora przez stałą oraz do wyliczania iloczynu skalarnego:

```
class wektor {
    //…
    friend wektor operator-(const wektor &v); // zmiana znaku
    friend wektor operator+(const wektor &x, const wektor &y);
    friend wektor operator-(const wektor &x, const wektor &y);
    friend wektor operator*(const wektor &v, double d);
    friend wektor operator*(double d, const wektor &v);
    // iloczyn skalarny x*y
    friend double operator*(const wektor &x, const wektor &y);
    wektor& operator+=(const wektor &v);
    wektor& operator-=(const wektor &v);
    wektor& operator*=(double d);
    //…
};
```

W klasie `macierz` zdefiniuj operatory indeksowania, zwracające odpowiednio wektor dla stałych macierzy i referencję do wektora w przypadku macierzy modyfikowalnych. Gdy indeks będzie miał wartość spoza dopuszczalnego zakresu zgłoś wyjątek. Nie zapomnij o zaprzyjaźnionych operatorach strumieniowych do czytania >> i pisania << macierzy. Zdefiniuj także operatory arytmetyczne dla dodawania i odejmowania macierzy, mnożenia macierzy przez stałą oraz do wyliczania iloczynu macierzy:

```
class macierz {
    //…
    friend macierz operator-(const macierz &m); // zmiana znaku
    friend macierz operator+(const macierz &p, const macierz &q);
    friend macierz operator-(const macierz &p, const macierz &q);
    friend macierz operator*(const macierz &m, double d);
    friend macierz operator*(double d, const macierz &m);
```

```
    friend double operator*(const macierz &p, const macierz &q);
    friend macierz operator~(const macierz &m); // transpozycja
    vect& operator+=(const macierz &m);
    vect& operator-=(const macierz &m);
    vect& operator*=(double d);
    //…
};
```

Uzupełnij definicję wektora o konstruktor, który umożliwi inicjalizację wektora w oparciu o listę wartości `initializer_list<double>`. Podobnie dla macierzy lista wektorów `initializer_list<wektor>` ma ułatwić inicjalizację. Przykładowe zastosowanie takich mechanizmów inicjalizacji to:

```
wektor w {1.62, 2.72, 3.14};
macierz m {
    {0, 1},
    {1, 1}
};
```

Wyjątki, które będziesz zgłaszać w definicjach obu klas zdefiniuj samodzielnie w postaci hierarchii klas dziedziczących po `logic_error`.

Na koniec napisz program, który rzetelnie przetestuje klasę `wektor` i `macierz` pod kątem inicjalizacji, kopiowania i operacji arytmetycznych.

Ważne elementy programu

- Użycie przestrzeni nazw `obliczenia`.
- Konstruktory i destruktory w klasach `wektor` i `macierz`.
- Implementacja semantyki kopiowania w tablicy bitów.
- Operatory indeksowania.
- Operatory dodawania, odejmowania i mnożenia dla wektorów i macierzy.
- Zgłaszanie wyjątków w konstruktorach i funkcjach składowych.
- W funkcji `main()` należy przetestować całą funkcjonalność wektorów i macierzy.