Zadanie 5

Otwarto: poniedziałek, 2 grudnia 2024, 00:00
Wymagane do: niedziela, 29 grudnia 2024, 23:59

Skoroszyt

Wstęp

Studenci powinni poznać:

- poziomy odporności na wyjątki;
- schematy pozwalające zapewnić co najmniej silną odporność na wyjątki;
- zarządzanie pamięcią z użyciem sprytnych wskaźników;
- schemat implementowania semantyki kopiowania przy modyfikowaniu.

Polecenie

W celu ułatwienia sobie nauki do egzaminów grupa studentów informatyki postanowiła usystematyzować swój sposób gromadzenia notatek za pomocą skoroszytu (ang. binder). Aby ta metoda była praktyczna i rzeczywiście usystematyzowana, studenci uznali, że skoroszyty, których będą używać, muszą spełniać pewne podstawowe wymagania.

Format treści notatek nie powinien być z góry narzucony (w końcu jedni studenci ręcznie notują na zajęciach, podczas gdy inni wolą wykorzystywać w tym celu sprzęt elektroniczny), jednak spójny w obrębie pojedynczego skoroszytu. Dodatkowo każda notatka powinna być widocznie oznaczona za pomocą (unikalnej w obrębie pojedynczego skoroszytu) zakładki, aby ułatwić jej wyszukiwanie.

Należy zaimplementować szablon klasy `binder` dostępny w przestrzeni nazw `cxx` o następującej deklaracji:

```
namespace cxx {
  template <typename K, typename V> class binder;
}
```

Gdzie typ klucza (zakładki) `K` ma semantykę wartości, czyli dostępne są dla niego bezparametrowy konstruktor domyślny, konstruktor kopiujący, konstruktor przenoszący, operatory przypisania i destruktor. Na typie `K` zdefiniowany jest porządek liniowy i można na obiektach tego typu wykonywać wszelkie porównania. O typie `V` (treści notatki) można jedynie założyć, że ma publiczny konstruktor kopiujący i publiczny destruktor.

Studenci chętnie dzielą się notatkami. Z racji tego, że są generalnie oszczędni preferują raczej wspólne używanie tego samego skoroszytu niż tworzenie nowej fizycznej kopii. Sytuacja zmienia się jednak, kiedy jeden z użytkowników współdzielonego skoroszytu postanawia wprowadzić w nim zmiany. W tej sytuacji, aby nie utrudniać nauki kolegom lub koleżankom, student musi udać się do ksero i stworzyć własną kopię skoroszytu.

Formalnie używany kontener powinien realizować semantykę kopiowania przy modyfikowaniu (ang. *copy on write*).

Kopiowanie przy modyfikowaniu to technika optymalizacji szeroko stosowana m.in. w strukturach danych z biblioteki Qt oraz dawniej w implementacjach `std::string`. Podstawowa jej idea jest taka, że gdy tworzymy kopię obiektu (w C++ za pomocą konstruktora kopiującego lub operatora przypisania), to współdzieli ona wszystkie wewnętrzne zasoby (które mogą być przechowywane w oddzielnym obiekcie na stercie) z obiektem źródłowym. Taki stan trwa do momentu, w którym jedna z kopii musi zostać zmodyfikowana. Wtedy modyfikowany obiekt tworzy własną kopię zasobów, na których wykonuje modyfikację.

Aby umożliwić efektywne zbieranie i modyfikowanie notatek, klasa `binder` powinna udostępniać niżej opisane operacje. Przy każdej operacji podana jest jej złożoność czasowa przy założeniu, że nie trzeba wykonać kopii. Złożoność czasowa kopiowania skoroszytu to $\mathcal{O}(n \log n)$, gdzie n oznacza liczbę przechowywanych notatek. Wszystkie operacje muszą zapewniać co najmniej silną gwarancję odporności na wyjątki, a tam gdzie to jest możliwe i pożądane (na przykład

?

konstruktor przenoszący i destruktor), nie mogą zgłaszać wyjątków.

- Konstruktory: bezparametrowy tworzący pusty skoroszyt, kopiujący i przenoszący. Złożoność $\mathcal{O}(1)$.

```
binder();
binder(binder const &);
binder(binder &&);
```

- Operator przypisania, który przyjmuje argument przez wartość. Złożoność $\mathcal{O}(1)$ plus czas niszczenia nadpisywanego obiektu.

```
binder & operator=(binder);
```

- Metoda `insert_front` wstawia na początek skoroszytu notatkę z podaną zakładką. Aby zachować unikalność zakładek, nie można wstawić nowej notatki z zakładką używaną aktualnie w skoroszycie – w takiej sytuacji metoda podnosi wyjątek `std::invalid_argument`. Złożoność $\mathcal{O}(\log n)$.

```
void insert_front(K const &k, V const &v);
```

- Metoda `insert_after` pozwala umieścić notatkę z podaną zakładką `k` bezpośrednio za notatką z zakładką `prev_k`. Metoda działa podobnie jak `insert_front`, jednak podnosi wyjątek `std::invalid_argument` również w sytuacji, gdy zakładki `prev_k` nie można znaleźć w skoroszycie. Złożoność $\mathcal{O}(\log n)$.

```
void insert_after(K const &prev_k, K const &k, V const &v);
```

- Bezparametrowa metoda `remove` usuwa pierwszą notatkę ze skoroszytu. Jeśli skoroszyt jest pusty, to podnosi wyjątek `std::invalid_argument`. Złożoność $\mathcal{O}(\log n)$.

```
void remove();
```

- Jednoparametrowa metoda `remove` usuwa ze skoroszytu notatkę z podaną zakładką. Jeśli takiej notatki nie ma w skoroszycie, to podnosi wyjątek `std::invalid_argument`. Złożoność $\mathcal{O}(\log n)$.

```
void remove(K const &);
```

- Jednoparametrowe metody `read` zwracają referencję do notatki, oznaczonej podaną zakładką. W wersji nie-`const` zwrócona referencja powinna umożliwiać modyfikowanie notatki. Operacja modyfikująca skoroszyt może unieważnić zwróconą referencję. Jeśli notatki z podaną zakładką nie ma w skoroszycie, metoda podnosi wyjątek `std::invalid_argument`. Złożoność $\mathcal{O}(\log n)$.

```
V & read(K const &);
V const & read(K const &) const;
```

- Metoda `size` zwraca liczbę notatek w skoroszycie. Złożoność $\mathcal{O}(1)$.

```
size_t size() const;
```

- Metoda `clear` opróżnia skoroszyt (przygotowując go na następny semestr), czyli usuwa z niego wszystkie notatki. Złożoność $\mathcal{O}(n)$.

```
void clear();
```

Dodatkowo, aby umożliwić szybkie przeczytanie wszystkich notatek (na przykład w ramach powtarzania przed egzaminem), skoroszyt powinien umożliwić efektywne przekartkowanie go, to znaczy udostępnić:

- Iterator `const_iterator` i metody `cbegin`, `cend`, oraz działające na iteratorze operatory przypisania (`=`), porównania (`==` i `!=`), inkrementacji (zarówno prefiksowy, jak i postfiksowy operator `++`), dostępu do wartości (`*` i `->`) pozwalające przeglądać notatki w kolejności ich występowania w skoroszycie. Jeśli nie nastąpiło skopiowanie skoroszytu, iterator pozostaje ważny. Iterator musi spełniać koncept `std::forward_iterator`. Wszelkie operacje w czasie $\mathcal{O}(1)$. Iterator służy jedynie do przeglądania notatek i za jego pomocą nie można modyfikować skoroszytu, więc zachowuje się jak `const_iterator` z biblioteki standardowej.

Kończąc uzgadnianie spójnego formatu skoroszytów, studenci uznali, że nie musi on udostępniać innych metod niż wyżej wymienione (jak na przykład wyszukiwanie zakładek niepodnoszące wyjątków – najwyżej kolega bez wiedzy o nich będzie miał utrudniony dostęp do szybkiego przeszukiwania tworzonego w trudzie skoroszytu). Waszym zadaniem jest pomóc tej grupie studentów poprzez stworzenie implementacji skoroszytów spełniającej wszystkie wymienione cechy.

?

Tam gdzie jest to możliwe i uzasadnione, należy opatrzyć metody kwalifikatorami `const` i `noexcept`.

Obiekt klasy `binder` powinien przechowywać tylko po jednej kopii wstawionych zakładek i notatek.

Klasa `binder` powinna być przezroczysta na wyjątki, czyli powinna przepuszczać wszelkie wyjątki zgłaszane przez wywoływane przez nią funkcje i przez operacje na jej składowych, a obserwowalny stan obiektu nie powinien się zmieniać. W szczególności operacje modyfikujące zakończone niepowodzeniem nie mogą unieważniać iteratorów.

Rozwiązanie będzie kompilowane poleceniem

```
g++ -Wall -Wextra -O2 -std=c++20 *.cpp
```

Rozwiązanie powinno być zawarte w pliku `binder.h`, który należy wstawić w Moodle.

Ocenianie rozwiązania

Ocena z testów automatycznych

Przyznawany jest jeden punkt za przejście wszystkich testów z każdej z sześciu grup testów. Zarządzanie pamięcią będzie sprawdzane poleceniem

```
valgrind --error-exitcode=123 --leak-check=full --show-leak-kinds=all --errors-for-leak-kinds=all --run-cxx-freeres=yes -q
```

Za błędną nazwę pliku zostanie odjęty 1 punkt. Za ostrzeżenia wypisywane przez kompilator zostanie odjęty 1 punkt. Nie ma punktów ułamkowych.

Ocena jakości tekstu źródłowego

Ocena jakości kodu jest z przedziału od 0 do 4 punktów. Nie ma punktów ułamkowych. Odejmujemy punkty za:

- brzydkie formatowanie kodu;
- niedotrzymanie wymaganej złożoności czasowej;
- nieprzejrzysty kod – sprawdzający nie mógł łatwo zrozumieć, jaką złożoność ma dana operacja, dlaczego implementacja poszczególnych metod zapewnia wymaganą gwarancję odporności na wyjątki, dlaczego nie cieknie pamięć, ewentualnie gdzie cieknie pamięć;
- nieprawidłowe oznaczenie lub brak oznaczenia metod jako `noexcept`;
- zaśmiecanie globalnej przestrzeni nazw, nieukrywanie funkcji i struktur pomocniczych jako prywatnych w implementowanej klasie;
- zbędne lub nadmiarowe `#include` (nawet gdy plik się kompiluje);
- niezastosowanie się do kryteriów oceniania poprzednich zadań;
- niezastosowanie się do uwag udzielonych przy ocenie poprzednich zadań.

O ile nie zostanie to wykryte przez testy automatyczne, to będą też odejmowane punkty za:

- brak `header guard`;
- braki `const`;
- nieefektywną implementację kopiowania przy modyfikowaniu, np. niepotrzebne lub zbyt częste kopiowanie;
- jawne użycie operatora `new`, np. zamiast użycia `std::make_shared`.

Przykłady użycia

Przykłady użycia znajdują się w pliku `binder_example.cpp`.

binder_example.cpp	28 listopada 2024, 16:39
binder_test_extern.cpp	2 stycznia 2025, 17:58
binder_test.cpp	3 stycznia 2025, 17:35