Zadanie 7

Otwarto: poniedziałek, 13 stycznia 2025, 00:00
Wymagane do: niedziela, 26 stycznia 2025, 23:59

Listy funkcyjne

Wstęp

Studenci powinni poznać:

- składnię wyrażeń lambda
- typ `std::function`
- podstawy programowania funkcyjnego z użyciem C++

Polecenie

Celem zadania jest zaimplementowanie operacji na listach, które niejawnie pamiętają ciąg elementów $[x_0, \ldots, x_{n-1}]$. Od typu `L` obiektu reprezentującego taką listę wymagamy jedynie, aby miał szablon metody o sygnaturze

```
template <typename F, typename A> A operator()(F f, A a);
```

gdzie typ `F` ma szablon metody o sygnaturze

```
template <typename X> A operator()(X x, A a);
```

Jedynym sposobem operowania na liście l jest jej uruchomienie jako funkcji z odpowiednią funkcją f i tak zwanym akumulatorem a. Dla obiektów l, f, a odpowiednio klas `L`, `F`, `A` oraz obiektów $x_0, x_1, \ldots, x_{n-1}$ odpowiednio klas $X_0, X_1, \ldots, X_{n-1}$ spełniona jest równość $l(f, a) = f(x_0, f(x_1, \ldots, f(x_{n-1}, a), \ldots))$, gdzie wywołania funkcji f po prawej stronie są odpowiednimi specjalizacjami operatora `()` dla typu `F`.

Operacje jako obiekty

Wszelkie wymienione tutaj operacje powinny być obiektami stałymi z odpowiednio przeciążonym szablonem metody `operator()`, tak aby umożliwić ich ewentualne przekazywanie jako argument funkcji.

- `auto empty`: funkcja stała reprezentująca listę pustą
- `auto cons(auto x, auto l)`: funkcja zwracająca listę `l` z dodanym na jej początek `x`
- `auto create(auto...)`: funkcja zwracająca listę składającą się z podanych argumentów
- `auto of_range(auto r)`: funkcja zwracająca listę powstałą z elementów `r`; można założyć, że `r` jest typu spełniającego koncept `std::ranges::bidirectional_range` ewentualnie opakowanego w `std::reference_wrapper`
- `auto concat(auto l, auto k)`: funkcja zwracająca listę powstałą z połączenia list `l` i `k`
- `auto rev(auto l)`: funkcja zwracająca listę z odwróconą kolejnością elementów listy `l`
- `auto map(auto m, auto l)`: funkcja zwracająca listę powstałą z listy `l` w taki sposób, że każdy jej element `x` zamieniany jest na `m(x)`
- `auto filter(auto p, auto l)`: funkcja zwracająca listę powstałą z listy `l` poprzez zostawienie tylko takich elementów `x`, które spełniają predykat `p(x)`
- `auto flatten(auto l)`: funkcja zwracająca listę powstałą z połączenia list pamiętanych w liście list `l`
- `std::string as_string(const auto& l)`: funkcja zwracająca reprezentację listy `l` jako `std::string` przy założeniu, że dla każdego elementu listy `x` działa `os << x`, gdzie `os` jest obiektem pochodnym `basic_ostream`; patrz przykłady użycia

Definicje wszystkich powyższych obiektów powinny znaleźć się w przestrzeni nazw `flist`.

Założenia

Przy wszystkich operacjach tworzących listy (czyli wszystkie poza `as_string`) argumenty przekazujemy przez wartość, co

?

oznacza robienie pełnej kopii. Użytkownik biblioteki, aby temu zapobiec, może użyć funkcji `std::ref` do opakowania argumentu. Ta powszechna praktyka pozwala jawnie decydować, co jest przekazywane przez wartość, a co przez referencję i pozwala uniknąć skomplikowanej składni *perfect forwarding*.

Przy uruchamianiu listy l poprzez wywołanie $l(f, a)$ zakładamy, że funkcja f zwraca obiekt takiego samego typu jak typ obiektu a. Jeżeli tak nie jest, to działanie wywołania $l(f, a)$ jest niezdefiniowane.

Wskazówki

Zadanie wymaga myślenia funkcyjnego. Przyjmijmy symboliczną notację $\lambda x \mapsto y$, która oznacza funkcję z argumentem x, a y jest wyrażeniem opisującym wynik i używającym x. Implementacja większości operacji będzie wyglądała tak: $\lambda(f, a) \mapsto y$, gdzie y będzie kodem z wyrażeniami zawierającymi f i a. Na przykład hipotetyczne wstawienie elementu x na koniec listy l mogłoby wyglądać tak:

$$h(x, l) = \lambda(f, a) \mapsto l(f, f(x, a)).$$

Wtedy implementacja `rev`(l) to po prostu $l(h,$`empty`$)$, lecz to może nie zadziałać, gdyż funkcja h nie spełnia założenia, że typ wyniku jest taki sam jak typ akumulatora. Wskazówka jest taka, aby zaimplementować `rev`(l) jako $\lambda(f, a) \mapsto y$, gdzie w wyrażeniu y uruchamiamy $l(\cdot, \cdot)$ z typem akumulatora `std::function<A(A)>`, gdzie `A` jest typem a.

Dodatkowe wymagania

Jeżeli mamy wybór między zdefiniowaniem własnej struktury a użyciem lambdy, należy użyć lambdy.

Pomocnicze definicje należy ukryć przed światem. W związku z tym, że rozwiązanie jest w pliku nagłówkowym, definicje ukrywamy w przestrzeni nazw na przykład o nazwie `flist::detail`.

Wymagania formalne

Rozwiązanie należy umieścić w pliku `funclist.h`, który należy wstawić do Moodle. Rozwiązanie będzie kompilowane na maszynie students poleceniem

```
clang++ -Wall -Wextra -std=c++23 -O2 *.cpp
```

Będzie używany kompilator `/opt/llvm/19.1.4/bin/clang++`.

Ocenianie rozwiązania

Ocena automatyczna

Za testy automatyczne zostanie przyznana ocena z przedziału od 0 do 6 punków. Za błędną nazwę pliku zostanie odjęty 1 punkt. Za ostrzeżenia wypisywane przez kompilator zostanie odjęty 1 punkt. Nie ma punktów ułamkowych.

Ocena jakości kodu

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

- nieelegancki styl, złe formatowanie kodu, brak komentarzy;
- rozwiązanie niefunkcyjne;
- tworzenie własnych struktur, jeżeli da się to samo zrobić z użyciem lambd;
- kod, którego sprawdzający nie jest w stanie zrozumieć;
- anonimową przestrzeń nazw w pliku nagłówkowym, zaśmiecanie przestrzeni nazw, np. przez użycie `using namespace std` w pliku nagłówkowym;
- wszystkie błędy, które nie powinny się już pojawiać po poprzednich zajęciach, np. brak *header guard*, przekazywanie dużych obiektów przez wartość, braki `const` itp., o ile nie zostały wykryte przez testy automatyczne.

Przykłady użycia

Przykłady użycia znajdują się w pliku `funclist_test_1.cpp`.

funclist_test_1.cpp	13 stycznia 2025, 00:05
funclist_test_extern.cpp	30 stycznia 2025, 12:21
funclist_test.cpp	30 stycznia 2025, 12:21