Zadanie 4

Otwarto: poniedziałek, 18 listopada 2024, 00:00
Wymagane do: niedziela, 8 grudnia 2024, 23:59

Wielomiany

Wstęp

Studenci powinni poznać:

- szablony oraz ich specjalizację i częściową specjalizację;
- techniki manipulowania typami i danymi na etapie kompilowania programu;
- poznać `constexpr` i dowiedzieć się, że w kolejnych standardach coraz więcej elementów z biblioteki standardowej ma takie oznaczenie, np. `std::max` od C++14;
- techniki radzenia sobie z szablonami o zmiennej liczbie argumentów;
- elementy biblioteki standardowej pomagające w powyższym, np. `type_traits`;
- poznać ideę perfect forwarding (użycie &&, std::forward);
- niektóre typy i funkcje biblioteki standardowej służące do metaprogramowania, np. `std::conditional`, `std::enable_if`;
- poznać mechanizm dedukcji argumentów klasy z szablonami wraz z definiowaniem własnych wskazówek dedukcji (od C++17);
- podstawowe informacje o konceptach;
- możliwości przeciążania funkcji, operatorów i konstruktorów.

Polecenie

Celem tego zadania jest zaimplementowanie obsługi pierścienia wielomianów wielu zmiennych. Należy zaimplementować następujący szablon klasy:

```
template <typename T, std::size_t N = 0> class poly;
```

Obiekt tej klasy reprezentuje wielomian jednej zmiennej $a_0 + a_1x + a_2x^2 + \ldots + a_{N-1}x^{N-1}$, gdzie współczynniki a_i są typu `T`. Liczbę `N` nazywamy *rozmiarem* wielomianu (żeby nie mówić stopień plus jeden). Typ `T` powinien być pierścieniem, tzn. powinny być na nim zdefiniowane binarne operacje `+`, `-`, `*` i unarna `-`, a domyślny konstruktor powinien dawać wartość odpowiadającą zeru.

Aby reprezentować wielomiany wielu zmiennych, typ `T` powinien być znowu wielomianem, czyli być typu `poly`. W ogólności do reprezentowania wielomianu m zmiennych używamy klasy `poly` rekurencyjnie do głębokości m, a końcowy typ `T` już nie jest typu `poly`, np.

```
poly<poly<poly<int, 3>, 2>, 4>
```

reprezentuje wielomian trzech zmiennych nad typem `int`. O wielomianie *m* zmiennych myślimy jak o funkcji zmiennych x_1, ..., x_m w następujący sposób. Najbardziej zewnętrzna definicja jest wielomianem zmiennej x_1 o współczynnikach będącymi wielomianami zmiennych od x_2 do x_m. Typ współczynnika jest wielomianem zmiennej x_2 ze współczynnikami będącymi wielomianami zmiennych od x_3 do x_m itd.

Konstruktory

Należy zaimplementować poniższe konstruktory klasy `poly`.

- Konstruktor bezargumentowy tworzy wielomian tożsamościowo równy zeru.
- Konstruktor kopiujący bądź przenoszący (jednoargumentowe), których argument jest odpowiednio typu `const poly<U, M>&` bądź `poly<U, M>&&`, gdzie `M <= N`, a typ `U` jest konwertowalny do typu `T`.
- Konstruktor konwertujący (jednoargumentowy) o argumencie typu konwertowalnego do typu `T` tworzy wielomian

?

rozmiaru 1.
- Konstruktor wieloargumentowy (dwa lub więcej argumentów) tworzy wielomian o współczynnikach takich jak wartości kolejnych argumentów. Liczba argumentów powinna być nie większa niż rozmiar wielomianu `N`, a typ każdego argumentu powinien być r-referencją do typu konwertowalnego do typu `T`. Wymagamy użycia „perfect forwarding", patrz `std::forward`.

Należy zapoznać się z szablonem `std::is_convertible` i konceptem `std::convertible_to`.

Powyższe konstruktory nie umożliwiają stworzenia wielomianu rozmiaru jeden (czyli stałego), którego jedyny współczynnik jest wielomianem. Dlatego należy zaimplementować funkcję `const_poly`, której argumentem jest wielomian p (obiekt typu `poly`) i która zwraca wielomian rozmiaru jeden, którego jedyny współczynnik to p.

Dodatkowo należy napisać odpowiednie wskazówki dedukcji typów (ang. *deduction guides*), tak aby można było tworzyć obiekty typu `poly` bez podawania argumentów szablonu (patrz przykład).

Operatory przypisania

Należy zaimplementować operatory przypisania kopiujący i przenoszący, których argument jest odpowiednio typu `const poly<U, M>&` bądź `poly<U, M>&&`, gdzie `M <= N`, a typ `U` jest konwertowalny do typu `T`.

Operatory arytmetyczne

Należy zaimplementować operatory `+=`, `-=`, `*=`. Argumentem operatorów `+=` i `-=` może być wielomian i argument ten jest wtedy typu `const poly<U, M>&`, gdzie `M <= N`, a typ `U` jest konwertowalne do typu `T`. Argumentem wszystkich trzech operatorów może być też stała referencja do wartości typu, który jest konwertowalny do typu `T`.

Należy zaimplementować operatory `+`, `-`, `*` w wersji binarnej, a operator `-` dodatkowo w wersji unarnej. W wersji binarnej powinny być możliwe trzy wersje użycia tych operatorów wymuszone odpowiednim konceptem:

- tylko lewy argument jest wielomianem,
- tylko prawy argument jest wielomianem,
- oba argumenty są wielomianami.

Typ wyniku powinien być najmniejszym typem zawsze mieszczącym wielomian wynikowy i możliwy do wydedukowania podczas kompilowania. W przypadku dodawania lub odejmowania rozmiar wyniku ze względu na daną zmienną powinien być maksimum rozmiarów argumentów. W przypadku mnożenia wielomianów o rozmiarach `N` i `M` ze względu na daną zmienną rozmiar wyniku powinien być `N + M - 1`, gdy `N` i `M` są niezerowe, a zero, gdy któryś z rozmiarów jest zerem.

Należy zaimplementować specjalizację szablonu `std::common_type` i pomocniczego typu `std::common_type_t`. Jeżeli oznaczymy typy współczynników wielomianów przez `U` i `V`, to typ współczynnika wielomianu wynikowego powinien być `std::common_type_t<U, V>`.

Operator indeksujący

Należy zaimplementować `operator[](std::size_t i)` w wersji zwykłej i `const` zwracający referencję do wartości współczynnika a_i wielomianu.

Metoda at

Należy zaimplementować dwie wersje metody `at` wyliczającej wartość wielomianu.

Pierwsza wersja metody `at` aplikuje argumenty do wielomianu jako funkcji wielu zmiennych. Liczba argumentów k może być różna od liczby zmiennych n. Wynika to z tego, że wielomian n zmiennych może być traktowany jako wielomian k zmiennych, gdzie współczynniki są wielomianami rozmiaru $\max(n-k, 0)$. Dopuszczamy zatem $k = 0$ lub $k > n$. W pierwszym przypadku wynikiem jest oryginalny wielomian. W drugim przypadku nadmiarowe argumenty są ignorowane, gdyż zmienne x_i dla $i > n$ po prostu nie występują w wielomianie.

Druga wersja metody `at` ma argument typu `const std::array<U, K>&`. Wtedy, jeżeli zawartość argumentu to `{v_1, ..., v_k}`, wywołanie jest równoważne wywołaniu `at(v_1, ..., v_k)`.

Argumenty `at` mogą być dowolnych typów, dla których daje się wyliczyć wartość wielomianu. W szczególności mogą to też być wielomiany.

Metoda size

Należy zaimplementować metodę `size`, której wynikiem jest rozmiar wielomianu.

?

Funkcja cross

Należy zaimplementować dwuargumentową funkcję `cross` realizującą mnożenie wielomianów według wzoru:

$$cross(p, q)(x_1, \ldots, x_n, y_1, \ldots, y_m) = p(x_1, \ldots, x_n)q(y_1, \ldots, y_m),$$

gdzie p i q to wielomiany odpowiednio n i m zmiennych, a wynikowy wielomian ma $n + m$ zmiennych.

Dodatkowe wymagania

- Wszystkie zdefiniowane konstruktory, metody czy funkcje powinny być `constexpr`, czyli powinna być możliwość użycia ich w trakcie kompilowania.
- Można założyć, że typ współczynnika `T`, jeśli nie jest to typ `poly`, udostępnia konstruktor domyślny tworzący element zerowy, konstruktor kopiujący i przenoszący, operator przypisania, operatory `+=`, `-=`, `*=`, dwuargumentowe operatory `+`, `-`, `*` oraz jednoargumentowy operator `-`.
- 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 `detail`.

Wymagania formalne

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

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

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;
- kod, którego sprawdzający nie jest w stanie zrozumieć;
- obliczenia w trakcie wykonywania programu;
- anonimową przestrzeń nazw w pliku nagłówkowym, zaśmiecanie przestrzeni nazw, np. przez użycie `using namespace std` w pliku nagłówkowym;
- tworzenie zbyt wielu wersji operatorów;
- 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 `poly_test_1.cpp`.

poly_test_1.cpp	16 listopada 2024, 16:54
poly_test_extern.cpp	12 grudnia 2024, 14:54
poly_test.cpp	13 grudnia 2024, 16:15