Duże zadanie, część 1

Wielomiany

Tegoroczne duże zadanie polega na zaimplementowaniu operacji na wielomianach rzadkich wielu zmiennych o współczynnikach całkowitych. Zmienne wielomianu oznaczamy x_0, x_1, x_2 itd. Definicja wielomianu jest rekurencyjna. Wielomian jest sumą jednomianów postaci px_0^n, gdzie n jest wykładnikiem tego jednomianu będącym nieujemną liczbą całkowitą, a p jest współczynnikiem, który jest wielomianem. Współczynnik w jednomianie zmiennej x_i jest sumą jednomianów zmiennej x_{i+1}. Rekurencja kończy się, gdy współczynnik jest liczbą (czyli wielomianem stałym), a nie sumą kolejnych jednomianów. Wykładniki jednomianów w każdej z rozważanych sum są parami różne. Wielomiany są rzadkie, co oznacza, że stopień wielomianu może być znacznie większy niż liczba składowych jednomianów.

Część 1 zadania

Jako pierwszą część zadania należy zaimplementować bibliotekę podstawowych operacji na wielomianach rzadkich wielu zmiennych. Opis funkcji znajduje się w pliku `poly.h` w formacie komentarzy dla programu `doxygen`.

Dostarczamy

W repozytorium `https://git.mimuw.edu.pl/IPP-login.git` (gdzie `login` to identyfikator używany do logowania w laboratorium komputerowym) znajduje się szablon implementacji rozwiązania tego zadania. Znajdują się tam następujące pliki:

- `src/poly.h` – deklaracja interfejsu biblioteki wraz z jej dokumentacją w formacie `doxygen`,
- `src/poly_example.c` – przykłady użycia biblioteki,
- `CMakeLists.txt` – plik konfiguracyjny programu `cmake`,
- `Doxyfile.in` – plik konfiguracyjny programu `doxygen`,
- `MainPage.dox` – strona główna dokumentacji w formacie `doxygen`.

Zastrzegamy sobie możliwość nanoszenia poprawek do tego szablonu. Będziemy je umieszczać gałęzi `template/part1`.

Wymagamy

Jako rozwiązanie części 1 zadania wymagamy:

- ewentualnego uzupełnienia implementacji lub rozszerzenia interfejsu biblioteki w pliku `src/poly.h`,
- stworzenia pliku `src/poly.c` z implementacją wymaganych funkcji,
- uzupełnienia dokumentacji w formacie `doxygen` tak, aby była przydatna dla programistów rozwijających moduł.

Powinna być możliwość skompilowania rozwiązania w dwóch wersjach: release i debug. Wersję release kompiluje się za pomocą sekwencji poleceń:

```
mkdir release
cd release
cmake ..
make
make doc
```

Wersję debug kompiluje się za pomocą sekwencji poleceń:

```
mkdir debug
cd debug
cmake -D CMAKE_BUILD_TYPE=Debug ..
make
make doc
```

W wyniku kompilacji odpowiednio w katalogu `release` lub `debug` powinien powstać plik wykonywalny `poly` oraz dokumentacja. W poleceniu `cmake` powinno być również możliwe jawne określenie wariantu release budowania pliku wynikowego:

```
cmake -D CMAKE_BUILD_TYPE=Release ..
```

Zawartość dostarczonych przez nas plików można modyfikować, o ile nie zmienia to interfejsu biblioteki i zachowuje wymagania podane w treści zadania, przy czym nie wolno usuwać opcji kompilacji: `-std=c11 -Wall -Wextra`. Zmiany mogą dotyczyć np. stylu, dokumentacji, deklaracji typedef, włączania plików nagłówkowych, implementacji funkcji jako `static inline`. Ewentualne dodatkowe pliki źródłowe, będące częścią rozwiązania, należy umieścić w katalogu `src`. Funkcja `main` programu musi znajdować się w pliku `src/poly_example.c`, ale zawartość tego pliku nie będzie oceniana w tej części zadania.

Dodatkowe wymagania i ustalenia

Rozwiązanie zadania powinno być napisane w języku C i korzystać z dynamicznie alokowanych struktur danych. Implementacja powinna być jak najefektywniejsza. Należy unikać zbędnego alokowania pamięci i kopiowania danych.

W interfejsie zostały przyjęte pewne konwencje, które mają ułatwić zarządzanie pamięcią. Dzięki tym konwencjom wiadomo, co jest właścicielem obiektu. Bycie właścicielem obiektu implikuje odpowiedzialność za zwolnienie używanej przez niego pamięci. W przypadku struktur `Poly` i `Mono` zwalnianie pamięci uzyskuje się poprzez wywołania odpowiednio funkcji `PolyDestroy` i `MonoDestroy`.

Podstawową konwencją jest *przekazywanie argumentów przez zmienną*. W języku C do tego celu użyty jest typ wskaźnikowy (np. `const Poly *`). Kod wołający funkcję, której przekazujemy argument przez zmienną, odpowiada za utworzenie odpowiedniego wskaźnika. Może to być wskaźnik na lokalną zmienną, bądź też wskaźnik uzyskany w wyniku alokacji pamięci na stercie (np. przez `malloc`). W tym drugim wypadku trzeba pamiętać, aby zwolnić tę pamięć. Odpowiedzialność za zwolnienie tak uzyskanej pamięci nigdy nie przechodzi na wołaną funkcję.

Przy niektórych funkcjach *argumenty przechodzą na własność* funkcji wołanej. Jest to zaznaczone w komentarzu opisującym daną funkcję. Funkcja przejmuje zawartość pamięci wskazywanej przez przekazany wskaźnik. Zazwyczaj jest to pojedyncza struktura (np. `Poly`, `Mono`) bądź tablica struktur.

Wynikiem niektórych funkcji jest struktura (np. `Poly`, `Mono`). Przyjmujemy tu konwencję otrzymywania *wyniku na własność*, co oznacza, że kod wołający taką funkcję otrzymuję zwracaną wartość na własność.

Przykłady przekazywania własności i zwalniania pamięci przez ostatniego właściciela:

```
{
  Poly p1 = ...
  PolyDestroy(&p1);
}
{
  Poly p1 = ...
  Mono m1 = MonoFromPoly(&p1, 7); // przekazanie własności p1
  MonoDestroy(&m1);
}
```

```
{
  Poly p1 = ...
  Mono m1 = MonoFromPoly(&p1, 7); // przekazanie własności p1
  Poly p2 = PolyAddMonos(1, &m1); // przekazanie własności m1
  PolyDestroy(&p2);
}
```

Obsługa błędów krytycznych

Jeśli wystąpi błąd krytyczny, np. zabraknie pamięci, program powinien zakończyć się awaryjnie kodem 1. Niezmienniki i warunki wstępne należy sprawdzać za pomocą asercji.