Konwersja z wyjątkami

Kontynuując przygody z konwersjami w różnych systemach liczenia, zajmiemy się tym razem systemem liczenia Majów w wersji „obliczeniowej”, tzn. używanej do liczenia (w odróżnieniu od reprezentowania dat). W zasadzie jest to po prostu dwudziestkowy system liczenia, ale lepiej brzmi „majański” system liczenia.

Do zaimplementowania – podobnie jak w zadaniu pierwszym – jest zestaw funkcji, które umożliwiają konwersję z łańcucha znaków na liczbę i z liczby na łańcuch cyfr majańskich (prototypy wszystkich funkcji są zdefiniowane w pliku nagłówkowym `mayan.h`)

Podstawowe informacje o systemie liczenia Majów są podane w Wikipedii (https://en.wikipedia.org/wiki/Maya_numerals)[1].

[1] Wybrałem wersję angielską, bo polska opisuje tylko system kalendarzowy, który ma ciekawą nieregularność, która zaciemni podstawowy element ćwiczenia.

Przykłady konwersji

Mimo atrakcyjnego sposobu zapisu używanego przez Majów, my pozostaniemy przy reprezentacji cyfr majańskich, zgodnej z konwencją z poprzedniego zadania. Tzn. pierwsza dziesiątka cyfr, to znaki ‘0’..’9’, natomiast druga dziesiątka cyfr to wielkie lub małe litery ‘A’..’J’ i ‘a’..’j’.

Kilka przykładów konwersji:

Argument funkcji	Wartość funkcji
`H`	`17`
`4A`	`90`
`1b7`	`627`
`1Dc0`	`13440`
`1kBC`	**`runtime_error`**
`JJJJJJJ`	**`runtime_error`**

Pozycja **`runtime_error`** w tabeli oznacza, że funkcja `mayan2uint` po napotkaniu błędu (nie-cyfra lub zbyt duża wartość) ma wygenerować (rzucić – `throw`) wyjątek takiego właśnie typu.

Organizacja projektu

Funkcje `mayan2uint` i `uint2mayan` (oraz funkcje pomocnicze) trzeba zaimplementować w pliku `mayan.cpp`. Deklaracje funkcji są umieszczone w pliku `mayan.h`.

Podstawowe znaczenie dla przygotowania testów będzie mieć organizacja pliku `zad_m4.cpp`, w którym trzeba będzie zapanować nad wyjątkami. Sytuacje wyjątkowe z pewnością trzeba sprawdzić, i warto, żeby pierwszy wyjątek nie powodował zakończenia testów lub co gorsza awarii programu.

Początek jest standardowy: tworzymy projekt, zmieniamy nazwę pliku źródłowego z `main.cpp` na `zad_m4.cpp`, Do folderu projektu kopiujemy plik `mayan.h` i dokładamy go do projektu (`Project > Add files…`). Tworzymy nowy plik `mayan.cpp` (`File > New`) i włączamy w nim nagłówek `mayan.h`.

Główne punkty

Najważniejsze elementy, które pojawią się w zadaniu, są następujące:

- generowanie wyjątku w sytuacjach błędnych (w funkcji `mayan2uint` i funkcjach pomocniczych),
- przechwytywanie wyjątku w programie testującym, co ważne, z możliwością dalszego wykonywania testów,
- liczenie wartości liczbowej, przy nie znanej z góry długości ciągu cyfr,
- wykrywanie nadmiaru przed wykonaniem operacji, która do nadmiaru może prowadzić.

Wygląda to całkiem poważnie, ale przy systematycznym podejściu do implementacji, nie powinno być kłopotów z realizacją.

Bardzo skromny start

Zaczniemy od bardzo skromnej zawartości w `mayan.cpp`:

```
#include "mayan.h"
#include <stdexcept>
#include <string>
using namespace std;

unsigned int mayan2uint(const std::string& digits)
{
    throw runtime_error("permanent exception!");
}
```

Z rzeczy nowych pojawia się tutaj plik nagłówkowy `stdexcept`, w którym znajdują się definicje standardowych klas wyjątków, m.in. `runtime_error`.

Dalsze kroki

W kolejnych krokach trzeba:

- przygotować pętlę testów dla sytuacji niepoprawnych „łatwych” (nie cyfry), w której brak wyjątku jest błędem, a przejęty wyjątek świadczy o poprawnym zachowaniu funkcji,
- zaimplementować funkcję wartości znaku, która generuje wyjątek dla znaku, który nie jest cyfrą,
- rozszerzyć `mayan2uint` tak, żeby łatwe testy niepoprawne przechodziły bez błędów,
- przygotować pętlę testów dla ciągów znaków, które dają poprawny wynik konwersji, generowanie wyjątku w sytuacjach błędnych (w funkcji `mayan2uint` i funkcjach pomocniczych),
- ponownie uzupełnić `mayan2uint`,
- dodać testy ze zbyt wielkimi wartościami i jeszcze raz poprawić `mayan2uint`.

Jeszcze nowości

W tym ostatnim punkcie użyjemy jeszcze jednego nowego pliku nagłówkowego: `limits`.

Dzięki niemu będziemy mieć dostęp do szablonu , z którego dowiemy się, jaka jest maksymalna wartość liczby całkowitej bez znaku:

```
numeric_limits<unsigned int>::max()
```

(Warto swoją drogą poświęcić chwilę czasu na zapoznanie się z różnościami dostępnymi w tym szablonie).

Jak można sobie poradzić z przewidywaniem nadmiaru?

Jeśli mam w planie wykonanie mnożenia przez `val`, to mogę podzielić maksymalną wartość przez `val` i sprawdzić, czy drugi czynnik nie jest aby większy od ilorazu. Jeśli tak – złapałem nadmiar, zanim zdążył wystąpić.

Jeszcze nowości (2)

To zadanie jest świetnym momentem na wprowadzenie ostatniego typu pętli, z którego możemy korzystać w C++ (`do – while`).

W odróżnieniu od instrukcji for i while, w których warunek jest sprawdzany przed wykonaniem powtarzanej instrukcji (lub bloku instrukcji), w pętli do warunek jest sprawdzany **po wykonaniu** (bloku) instrukcji. To oznacza, że instrukcja (blok) zostanie wykonana przynajmniej jeden raz.

```
do
{
   // powtarzany blok instrukcji
} while (warunek);
```

Dostarczanie rozwiązania

Rozwiązaniem zadania jest plik **`mayan.cpp`**.

Proszę złożyć w Moodle tylko ten jeden plik (oczywiście na początku pliku trzeba umieścić w komentarzu imię, nazwisko i nr albumu autora).

Rozwiązanie należy przekazać do:

9 listopada 2021 23:59.