Gevorderd Programmeren 2009-2010 (2de zit)

Examen Practicum

P. Kłosiewicz

6 september 2010

Inleiding

Brainfuck is een *esoterische* programmeertaal bedacht door Urban Müller rond 1993. De taal bestaat uit 8 commando's die de werking van een (single-tape) Turing machine nabootsen. Hierdoor is deze taal natuurlijk Turing-compleet en kan dus gebruikt worden om elke complexe bewerking te programmeren.

De taal omschrijft bewerkingen die worden uitgevoerd op een tape-geheugen dat bestaat uit cellen die elk 1 byte groot zijn. Over de tape schuift een lees/schrijf-kop die de bytes op de tape kan lezen en aanpassen (Het aanpassen van de tape-cellen wordt in Brainfuck beperkt tot increment en decrement van de byte waarde in een cel). Bovendien kan men in Brainfuck ook gebruik maken van (beperkte) input/output om gegevens uit een cel af te printen of het programma van user-input voorzien.

Naast het werkgeheugen in de vorm van een tape heeft een Brainfuck-machine ook een programma-geheugen waarin de opeenvolgende instructies worden bijgehouden.

De huidige plek in beide geheugens wordt aangeduid met respectievelijk een "data pointer" en een "instruction pointer". Dit zijn `unsigned int`s die bij de start van het programma beide 0 zijn (dit is logisch: tape-kop uiterst links en eerste instructie van het programma is de huidige instructie).

De tape wordt verondersteld nullen te bevatten bij de aanvang van de machine.

Zie de bijgevoegde wiki-pagina over Brainfuck voor een meer complete beschrijving.

De opgave

De opgave bestaat uit het schrijven van een object-georiënteerde Brainfuck-VM (virtuele machine) in C++ dat een Brainfuck programma uit een file kan inlezen en vervolgens uitvoeren. Indien het programma fouten bevat, moet de VM daar ook mee kunnen omgaan.

Implementatie

Voor een uitbreidbare en flexibele implementatie van deze VM zijn de volgende entiteiten (denk: classes) nodig:

- De **data** zijn de gegevens die op de tape kunnen worden opgeslagen en die door de instructies in een BF programma kunnen worden gemanipuleerd.
  Implementeer een class `Data` als container voor het Brainfuck data type `byte`. (Een `byte` is niets anders is dan een `typedef` voor een `unsigned char`)

- Een **instructie** kan de tape één cel opschuiven, een cel lezen of schrijven, de inhoud van een cel aanpassen of zorgen voor input/output.
  Je hebt dus een abstracte base class `Instruction` nodig met de volgende afgeleide classes voor de specifieke instructies:

  - `InstructionRight` (‘`>`’) die de leeskop van de tape een cel naar rechts opschuift. Dit betekent dat de “data pointer” met 1 wordt vergroot.
  - `InstructionLeft` (‘`<`’) die de leeskop van de tape een cel naar links opschuift. Dit betekent dat de “data pointer” met 1 wordt verkleind.
  - `InstructionInc` (‘`+`’) die de byte in de huidige tape-cel met één vergroot.
  - `InstructionDec` (‘`-`’) die de byte in de huidige tape-cel met één verkleint.
  - `InstructionOut` (‘`.`’) die de byte in de huidige tape-cel in ASCII formaat naar `std::cout` schrijft.
  - `InstructionIn` (‘`,`’) die de byte in de huidige tape-cel opvult met een `char` ingelezen vanuit `std::cin`.
  - `InstructionJumpFw` (‘`[`’): als de byte op de tape-cel aangeduid door de huidige data pointer nul is, spring dan (voorwaarts) naar de instructie volgende op de “matchende” `InstructionJumpBw`.
  - `InstructionJumpBw` (‘`]`’): als de byte op de tape-cel aangeduid door de huidige data pointer NIET nul is, spring dan (achterwaarts) naar de instructie volgende op de “matchende” `InstructionJumpFw`.

  Zorg ervoor dat elke instructie een virtuele `execute` method heeft die, gegeven de huidige data, de data pointer en de instruction pointer, de correcte actie uitvoert.

- Een **programma** is een lijst/vector/... van **instructies** waarbij de volgorde uiteraard van belang is. Schrijf daarom ook een klasse `Program` die een container van `Instruction`s bevat.
  De constructor van deze klasse moet via een `std::string` de file name van het in te lezen programma krijgen. Vervolgens leest de constructor de file karakter per karakter in en “maakt” de overeenkomende instructies. Vergeet niet de “matchende” `[` en `]` instructies te voorzien van hun instruction pointers (dit om te voorkomen dat je op “runtime” de matchende jumps moet zoeken). De voor Brainfuck onbekende karakters worden genegeerd.

- De **Brainfuck VM** zelf tenslotte, bevat een lineaire container van `Data` objecten (= een tape) en kan een `Program` runnen.

In de originele implementatie is de tape 30000 bytes lang. Doe dit ook in jouw code. De huidige toestand van de VM wordt bepaald door (1) de inhoud van de tape, (2) de data pointer (die de huidige cel onder de tape-kop aanduidt) en (3) de instruction pointer (die de huidig uitgevoerde instructie aanduidt).

Om een Brainfuck VM te gebruiken wordt eerst een programma ingelezen uit een file. Nadien wordt de machine gemaakt en opgestart met het programma voor uitvoering, hetgeen al dan niet succesvol eindigt. Zie voorbeeld:

Voorbeeld

De implementatie van de afzonderlijke klassen zou de volgende code in de **main**-functie moeten toestaan:

```
int main(int argc, char* argv[]) {
  try {
    cout << "Brainfuck VM ready for some serious action!" << endl;

    // Get program file name:
    if (argc != 2) {
      throw runtime_error("Usage: bfm sourcecode.bf");
    }
    string fileName(argv[1]);

    // Create machine and load program:
    BFMachine machine;
    Program prog(fileName);

    // Run program:
    machine.run(prog);
  } catch (exception& e) {
    cout << "Fatal error encountered:" << endl << e.what() << endl;
  }
  return 0;
}
```

Opmerkingen

Let op de volgende (zeer belangrijke) zaken:

- Implementeer "minimale" code om de structuur van de besproken entiteiten op te bouwen. Gebruik je tijd dus nuttig om eerst de gevraagde features te implementeren. Zaken die je niet hoeft te schrijven (bv. overbodige constructoren, door de compiler gesynthetiseerde methodes, andere niet gebruikte methodes, ...) laat je best achterwege.
- Gebruik exception handling voor fout-afhandeling. Je mag hierbij beroep doen op de class `std::runtime_error` uit `#include <stdexcept>` i.p.v. een eigen exception class structuur te schrijven.
  O.a. de volgende fouten moet je kunnen afhandelen (de foutboodschap mag als string in de `runtime_error` komen)
  - Problemen met laden van een program file.
  - Problemen met matchen van de `[` en `]` instructies.
  - *Instruction pointer out of bounds*: het programma heeft een illegale jump gemaakt.
  - *Data pointer out of bounds*: de tape is langs een van de twee kanten "afgerold".
- "Verpak" de Brainfuck machine classes in de `bfm` namespace.
- Gebruik de standard library waar nodig, nuttig en/of warm aanbevolen.
- Vergeet niet `const`-correct te zijn!
- Voorzie je code van duidelijke maar summiere commentaar en schrijf in **elk bestand je naam en rolnummer**.
- Test jouw implementatie op de bijgevoegde `.bf` files.

Veel succes!