

# Opdracht 4: $n$ -op-een-rij (recursie, make en grafische interfaces)

November 7, 2002

## 1 Inleiding

In deze opdracht leer je werken met recursie, Makefiles en Qt designer. Lees voor je aan de opdracht begint eerst de corresponderende hoofdstukken in deel II van het dictaat.

Concreet dien je in deze opdracht (1) een elegante oplossing te bedenken voor het spel  $n$ -op-een-rij op een  $r \times k$  bord, en (2) deze uitwerking grafisch weer te geven. De hierboven genoemde parameters  $n$ ,  $r$  en  $k$  worden door de speler ingesteld. Kiest de speler bijvoorbeeld  $n = 4$  en  $r = 5$  en  $k = 6$  dan is dit het spel ‘4-op-een-rij’ op een bord met 5 rijen en 6 kolommen. De regels van het spel zijn als volgt: spelers mogen om de beurt hun ‘schijf’ in een van de kolommen gooien. De schijf valt door tot deze de onderste rij heeft bereikt, of tot vlak voor een al eerder gespeelde ‘schijf’ (met andere woorden: de stenen vallen ‘naar beneden’). De winnaar van het spel is die speler die het eerst  $n$  van zijn eigen stenen horizontaal, verticaal of diagonaal op een rij heeft.

## 2 Opgaven

**Opgave 1** Schrijf een programma dat twee spelers om de beurt een zet laat doen en dat (1) alleen geldige zetten toestaat, en (2) na elke zet bepaalt of een van de spelers heeft gewonnen en zoja wie (de winnaar wordt afgedrukt).

Schrijf hiervoor een nieuwe klasse `bord` die in ieder geval een 2-dimensionaal array bevat waarin het bord is opgeslagen. Een element van de array geeft een veld van het bord weer. Dit element heeft de waarde 0, als speler 0 daar een schijf heeft staan, 1 als speler 1 daar een schijf heeft staan, en  $-1$  als niemand daar iets heeft staan. Geef `bord` in elk geval de volgende methoden (eventuele parameters en return waarden moet je zelf invullen):

1. `bord(...)`: de constructor.
2. `zet(...)`: doe een zet voor een speler in een bepaalde kolom.
3. `is_zet_geldig(...)`: controleer of een zet geldig is.
4. `is_bord_vol(...)`: geef aan of het bord vol is.
5. `winnaar(...)`: geef aan of er een winnaar is en zoja wie.
6. `maak_leeg(...)`: maak het bord leeg, zodat een nieuw spel kan beginnen.

Aanwijzingen:

- De methode `zet(...)` roept natuurlijk intern de functie `is_zet_geldig(...)` aan.
- Denk na voordat je een functie als `winnaar` schrijft: in principe moet je hiervoor de rijen, kolommen en diagonalen aflopen op zoek naar  $n$ -op-een-rij, maar je wilt deze functie wel het liefst zo efficiënt mogelijk maken. Heeft het bijvoorbeeld zin om nog verder horizontaal (bijvoorbeeld naar rechts) te controleren als de eerste schijf van een speler op  $n - 1$  velden van de rechter-zijkant ligt?

- Een methode als `is_bord_vol(...)` is op verschillende wijze te implementeren. Je kunt alle velden bekijken om te zien of er een schijf op staat, maar je kunt ook een eenvoudige teller bijhouden die na elke zet opgehoogd wordt. Het veld is dan vol als deze teller gelijk is geworden aan het aantal velden op het bord.

**Opgave 2** Wijzig het door jou geschreven programma zodanig dat het wordt opgesplitst in een bestand waarin `main()` staat, een bestand waarin de implementatie van klasse `bord` staat, en een header file met daarin de definitie van deze klasse (zonder implementatie).

Schrijf een `Makefile` voor deze files zodanig dat de twee `.cc` bestanden apart worden gecompileerd en uiteindelijk worden samengevoegd (“gelinkt”) tot een executeerbaar programma. Schrijf ook een `clean` regel.

**Opgave 3** Breid het door jou geschreven programma uit met een functie die, gegeven een bepaalde stand op het bord en de speler die aan de beurt is, recursief alle mogelijke vervolgzetten bepaalt. Deze functie dient af te drukken wanneer er een winnaar is (en wie) en ook wanneer het bord vol is zonder dat iemand heeft gewonnen (remise). Als een winnaar is gevonden, dan breekt daar de recursie natuurlijk af. Bepaal met dit programma het aantal mogelijke ‘eindstand-borden’ voor een spel waarbij  $n = r = k = 3$  (met andere woorden: bepaal het aantal mogelijke ‘partijen’). Deze eindstand-borden hoeven niet *per se* verschillend te zijn, maar wel afkomstig vanuit verschillende spelverlopen.

Aanwijzingen:

- Bestudeer paragraaf 3.3 uit deel II van het dictaat om te zien hoe zo’n recursieve functie wordt geschreven.
- Schrijf eerst het ‘skelet’ van je programma: welke functie ga je gebruiken, en waarvoor, welke ‘loops’ heb je waar nodig, etc. Vul dit raamwerk vervolgens in.
- Als je deze functionaliteit wilt testen, kies dan een klein bord, bijvoorbeeld  $3 \times 3$ . Bij grotere borden (bijvoorbeeld  $10 \times 10$ ) duurt het al snel vele eeuwen voordat je het hele bord recursief hebt doorgerekend.

**Opgave 4** Breid het programma uit zodanig dat tegen de computer gespeeld kan worden. Steeds als de gebruiker een zet heeft gedaan, wordt deze gevolgd door een (zo goed mogelijke) computer-zet. Het is aan de speler om te bepalen of hij/zij of de computer begint.

Aanwijzingen:

- Bekijk eerst aandachtig paragraaf 3.4 uit het dictaat. Een soortgelijke oplossing als daar wordt gegeven kun je ook hier gebruiken. Wanneer je een betere oplossing hebt bedacht (of een aanpassing), mag je die natuurlijk ook gebruiken.
- Schrijf eerst een functie die van twee zetten bepaalt welke de beste is, gegeven het aantal winst-, verlies-, en remise-potten die het gevolg (kunnen) zijn van die zetten.
- Gebruik een functie `bepaal_goede_ervolg_zet(...)`.

**Opgave 5** Schrijf een grafische interface voor het spel  $n$ -op-een-rij met behulp van Qt designer. Een speler moet het spel of tegen de computer of tegen een andere (menselijke) speler kunnen spelen. De computer controleert wanneer gewonnen is en door wie, etc. Geef de schijven van de ene speler een andere kleur en een ander opschrift (of plaatje) dan die van de andere speler. Het spel moet meerdere keren gespeeld worden.

Aanwijzingen:

- Bestudeer eerst aandachtig Hoofdstuk 5 uit deel II van het dictaat.
- Het opgeven van  $n$ ,  $r$  en  $k$  door de gebruiker hoeft niet *per se* via de interface gedaan worden (mag wel). Je zou deze waarden bijvoorbeeld als ‘command-line parameters’ mee kunnen geven.

### 3 Opmerkingen

Houd je functies compact. Het liefst geen functies die langer zijn dan een scherm aan programma-code. Voorzie elke functie van commentaar.

In te leveren: lever een schijfje met alle programma-code (*niet* de executables!) in, of stuur een email met je uitwerkingen als *attachment* naar `pm@liacs.nl`. Zorg er voor dat je *alleen* de programma-code inlevert en niet de executables. Je kunt ofwel alle bestanden apart ‘attachen’, of er (op UNIX) een enkel `tar` bestand maken (waarschijnlijk makkelijker). Met `tar` kun je vanaf een UNIX systeem meerdere bestanden in 1 file stoppen. Als bijvoorbeeld je bestanden staan in directory `./opdr4_dali/`, dan kun je deze directory, alle bestanden in deze directory, en alle sub-directories en hun bestanden in deze directory in 1 `tar` bestand (bijv. genaamd `opdr4_dali.tar`) stoppen door middel van het volgende commando: `tar cvf opdr4_dali.tar opdr4_dali/`. Nogmaals: zorg ervoor dat deze directories *alleen* programma-code bestanden bevatten. Stuur dit `tar` bestand als attachment naar `pm@liacs.nl`. Uiterste inleverdatum: 6/12/02, 17:00 uur. Verder ook een listing/afdruk op papier in de speciaal daarvoor bestemde doos ”Programmeermethoden” in de metalen boekenkast in zaal 301. Gaarne een envelop om het geheel heen doen. Overal datum en namen van de makers vermelden.