Academy Creative Technology

Opleiding Informatica

Taken Advanced C++

Ron Akkersdijk

2014 - 2015

Woord vooraf

De opdrachten mogen zowel solo of als duo worden uitgevoerd. Bij elke opdracht hoort naast de code uiteraard ook een klein rapportje waarin je uitlegt hoe je de oplossing hebt bedacht, hoe de oplossing functioneert, en hoe je het resultaat hebt getest (denk aan “mooi weer”, “slecht weer”).

Omdat de meeste compilers nog niet volledig C++11 compliant zijn moeten alle uitwerkingen C++03 compliant zijn. Het gebruik van C++11 features is dus (nog) niet toegestaan.

De eerste drie opdrachten (voorzien van jaartal 2014) zijn de *maken en inleveren* opdrachten voor deze module. De rest zijn ideeen voor wat *voor-je-zelf* oefeningen.

Beoordeling

De basisopdracht goed doen en een, fatsoenlijk, begrijpelijk, rapport levert een 7 op.
Meer features, een slimmer algorithme, een beter rapport, etc. levert extra punten op.

1. Over *Callbacks* en *Pointers* naar methodes (2014)

Techniek

In de slides wordt aangegeven dat *callbacks* zowel met java stijl “interface classes” kan (in C++ via abstracte classes en e.v.t. multiple inheritance) maar ook met het “pointer naar functie concept” gerealiseerd kunnen worden. Zo wordt als voorbeeld een generieke menu handler beschreven die geïmplementeerd is met behulp van pointers naar (member) functies. Op eerdere slides stond een voorbeeld van een programma met cursor besturing (class Screen) via het toetsenbord.

Context

De *emacs* editor biedt een groot scala aan ingebouwde “acties”, die via de “**`bind`**” actie gekoppeld kunnen worden aan een toetsaanslag. Zo zijn meestal de gewone karakters gekoppeld aan de actie “**`insertSelf`**” (de toetsaanslag wordt altijd als parameter meegegeven), terwijl de control toetsen meestal zijn gekoppeld aan bijzondere acties. De emacs editor gebruikt meerdere *binding* -tabellen zodat b.v. ^X^S (control-X, control-S), in feite afgehandeld gaat worden als (de ^X in de hoofdtabel) “schakel over naar 2e tabel”, en dan (de ^S in de tweede tabel) de actie “**`saveToFile`**”. Omdat sommige toetsen van je toetsenbord eigenlijk een hele reeks karakters opleveren (b.v. de *cursor-up* toets geeft meestal **escape [A**) zit er effectief een hele “boom” van *binding* tabellen in emacs.

Daarnaast kan emacs ook een reeks toetsaanslagen opslaan in een buffer, en vervolgens kan je die buffer dan weer in een van de tabellen opnemen als “macro-actie” (via de actie: “**`executeBuffer`**(*nummer*)”). Die buffers bevatten ook de bestanden die ge-edit worden, dus bestanden en macro's zijn daarmee in feite uitwisselbaar geworden. Omdat elke gebruiker zo zelf volledig kan kiezen welke toetsaanslag wordt gekoppeld aan welke actie, is emacs extreem aanpasbaar aan je persoonlijke smaak. In feite creeert de gebruiker hiermee zijn eigen, persoonlijke, user-interface. Omdat je vanuit emacs ook andere programma's kunt opstarten, is het mogelijk om van emacs een complete “desktop omgeving” te maken.

Opdracht

We willen een stukje van het gedrag van emacs nabouwen.

Bij het `MenuHandler` voorbeeld van de slides, is er één (1) tabel van acties, en welke actie wordt uitgevoerd in relatie tot de invoer, wordt bepaald door het volgnummer van die actie in die tabel.

Waar we naar toe willen is een flexibelere, emacs stijl, variant waarbij de gebruiker zelf, interactief, acties kan koppelen aan invoer. De `Applicatie` class zelf biedt nog steeds één verzameling `MenuFunctions` aan, maar die zullen in eerste instantie nog niet in het aangeboden menu voorkomen.

De herziene `MenuHandler` biedt de "`bind`" actie aan zodat de gebruiker een actie uit de applicatie verzameling kan koppellen aan bepaalde invoer. Op deze manier stelt de gebruiker zelf z'n eigen menu samen. Het kan dus zijn dat de applicatie meer acties aanbied dan dat wat "nu" daadwerkelijk via de *binding* tabellen bereikbaar is.

De `MenuHandler` heeft in eerste instantie één tabel die invoer koppelt aan uit te voeren acties (waar initieel uiteraard die "`bind`" in staat). De gebruiker moet daarnaast de mogelijkheid krijgen om, dynamisch, meer tabellen (m.a.w. sub-menu's) te introduceren, zodat een hierarchie van acties kan ontstaan. De `MenuHandler` biedt daar uiteraard weer passende acties voor aan.

Hou bij deze opdracht goed voor ogen dat we hier te maken hebben met 2 verzamelingen met acties. De acties die eigenlijk bij de applicatie horen (dat wat het programma doet), en de acties die bij de interface horen zoals de "`bind`". Hoe je hiermee omgaat in je C++ code is een van de uitdagingen van deze opdracht.

Omdat het bij deze opdracht vooral om het mechanisme gaat is het scala aan acties van de eigenlijke applicatie (class `Factory` op de slides) niet erg belangrijk. Een verzameling methodes "`actie1`", "`actie2`", etc, die alleen maar iets afdrukken, zodat je kan zien dat de goede actie wordt uitgevoerd, is goed genoeg. Maar niets belet je om daar iets mooiers van te maken.

Het zou mooi zijn als de gebruiker, weer via de `MenuHandler`, ook een "`saveMenu`" en "`loadMenu`" actie aangeboden zou krijgen zodat je niet elke keer alles opnieuw moet instellen als je je programma start, maar ook dit is een extraatje.

Wat je hier voor "de invoer" gebruikt is geheel naar eigen keuze, een getal, een letter, een woord, enz. Het leukste is uiteraard als je in emacs stijl "losse" karakters kan gebruiken (onder win32 mischien te doen via `conio.h`, onder unix/linux/macos te regelen via de "terminal settings" [zie ook losse bijlage **`screen.cc`**]). In die situtatie is de '`show`' methode van `MenuHandler` niet meer relevant geworden. Maar, dit is alleen maar een extra idee, gewoon een (sub) menu en "een opdracht" gevolgd door ENTER is ook prima.

Beslis zelf waar "pointer-to-function" of "pointer-to-member" de netste aanpak zou zijn dan wel waar een "interface class" beter is.

Wat aanvullende bronnen:

screen.cc = een mini programma: keyboard => cursor besturen (niet voor win32).
http://en.wikipedia.org/wiki/Emacs
http://en.wikipedia.org/wiki/ANSI_escape_code
http://en.wikipedia.org/wiki/Control_character

2. Generieke Algoritmen (2014)

Context

Op een wat achtergebleven onderwijsinstelling worden de gegevens van studenten nog bijgehouden in meerdere tekstbestanden (`csv` stijl o.i.d.). Per student worden, op een regel: naam, voornaam, woonplaats, geslacht en klas bijgehouden (maak zelf wat testbestanden). Het staat je vrij om een ander scheidingsteken te gebruiken dan de '`,`' van `csv`. Een '`|`' o.i.d. is ook prima. De tekstbestanden kunnen informatie van meerdere studenten bevatten, b.v.

een bestand voor elke klas of een bestand per woonplaats of volstrekt willekeurig, maar elke student staat dan wel op een aparte regel.

Om een beetje mee te tellen in deze tijd, wordt besloten over te gaan op een XML database.

Opdracht

Schrijf een programma dat begint met alle tekstbestanden in te lezen en de informatie in een container te stoppen als verzameling `Student` objecten. Omdat gegevens vaker voor kunnen komen, worden alle student objecten vervolgens gesorteerd op naam en worden duplicaten geëlimineerd (Bedenk zelf een definitie voor wat hier als “dubbel” telt). Vervolgens worden de `Student` objecten gesorteerd op klas en tenslotte worden ze weer als XML tekst weggeschreven naar een nieuw tekstbestand.

Maak zoveel mogelijk gebruik van *istream* en *ostream iteratoren*, *functie objecten* en in ieder geval ook de volgende generieke functies: `copy`, `sort`, `unique`, `stable_sort`, `for_each`.

Voetnoot

Bij de beoordeling van deze opdracht wordt vooral gelet op het correct en optimaal gebruik van C++ features zoals, *classes*, *inheritance*, *member initialization*, *const*, *reference*, *operatoren*, *templates*, *generieke algoritmes* en de optimale taakverdeling tussen classes, methodes en gewone functies. Een “java stijl” oplossing wordt hier zeker niet geaccepteerd.

3. Containers, iteratoren en operatoren (2014)

Opdracht

Realiseer een `SortedBinaryTree<T>` container template, inclusief bijpassende `SortedBinaryTree<T>::iterator` class. De relatieve rangschikking van elementen wordt bepaald door de `<` operator van het type `T`.

De boom hoeft niet gebalanceerd te zijn o.i.d., dus hoef je bij het toevoegen van elementen niet de volledige structuur te herzien zoals b.v. nodig zou kunnen zijn bij *balanced trees*. Bij het verwijderen van elementen kan het wel nodig zijn om kleinschalig de structuur iets te herzien. Het is dus potentieel mogelijk dat een instantie van zo'n boom, afhankelijk van welke inserts en deletes er gebeuren, uit louter linker takken bestaat, wat niet echt een ideale situatie is.

Als je met een “forward” iterator door de boom loopt dan moeten de elementen in de gesorteerde volgorde (m.a.w. in de *in-order* volgorde) bezocht worden.

De verzameling van aangeboden methodes, de signatuur daarvan, en het gedrag daarvan moet zoveel mogelijk analoog zijn aan dat van de bestaande containers uit de STL en met name die van `map<K,V>` en `set<V>` (die zijn immers zelf *RedBlackTrees*).

De minimale set aan features:

- Default Constructor
- optioneel: Copy Constructor van een andere `SortedBinaryTree<T>`.
- Destructor
- `empty()`
- `size()` telt de elementen
- `clear()` verwijdert alle elementen
- `insert(`*value*`)` voegt een item toe (tenzij die al bestaat, anders NOP)
- `erase(`*value*`)` verwijdert een item (tenzij die niet bestaat, anders NOP)
- optioneel: `count(`*value*`)`

- een *forward iterator*
- optioneel: een *reverse iterator*
- de `begin()` en `end()` voor iteratoren
- optioneel: `rbegin()` en `rend()`
- wenselijk: `find(`*value*`)`
- `erase(`*iterator*`)`

Voor de iterator(en) bestaat minstens:

- een `++x` operator (*walk forward*)
- optioneel: `--x` operator (*walk backwards*)
- een `const T& *x` operator (*dereference*)
  Het is niet de bedoeling dat je via deze route de inhoud van de boom zou kunnen veranderen, vandaar de '`const`'.
- de `==` en `!=` operator
  Dat de iteratoren die je vergelijkt mischien bij verschillende bomen behoren, wat een onzin resultaat zou opleveren, is een complicatie die je niet hoeft op te lossen.

Voetnoten

Je kan, naar eigen keuze, de boom realiseren als recursive boom, of als een front-end class die weer een aantal *`SBTNodes`*`<T>`'s beheert die de eigenlijke boom vormen.

Idem, het kan handig zijn om eerst een baseclass `BinaryTree<T>` te definieren, met `SortedBinaryTree<T>` als afgeleide daarvan (maar is misschien weer meer werk).

Als er een of meerdere iteratoren actief zijn en de structuur van een boom wordt veranderd vanwege een `insert` of `erase`, dan kan het zijn dat een *iterator* opeens in limbo verwijst of naar een element waarvan de inhoud is veranderd. Daar hoeft jouw implementatie geen rekening mee te houden (*Design by Contract* legt de verantwoordelijkheid hiervoor immers bij de “gebruiker”, lees: de applicatie).

Bijlage: **`Array.h`**

4. Include directives en *ostreams*

Context

Bij het reguliere C++ onderwijs werd vertelt hoe, met behulp van *include guards* (`#ifdef, #define, #endif`), de compilatie tijd van een programma bekort kan worden.

Opdracht

Schrijf een programma dat het volgende genereert:

a) `N` header files “`widget_i.h`” `(1 <= i <= N)`.
   Een header file definieert een (lege) class `widget_i` en bevat verder `M` (lege) commentaar regels.
b) `N` header files “`screen_i.h`” `(1 <= i <= N)`.
   Een header file include alle `N` widget header files (van a) en definieert een (lege) class `screen_i`.
c) `N` header files “`opt_screen_i.h`” `(1 <= i <= N)`.
   Een header file include alle `N` widget header files (van a), maar nu gebruik makend van include guards, en definieert een (lege) class `screen_i`.
d) Een hoofdprogramma `main1.cc` die alle screen header files (van b) include en verder een lege `main` bevat.

e) Een hoofdprogramma `main2.cc` die alle geoptimaliseerde screen header files (van c) include en verder een lege `main` bevat.
f) Optioneel: Een *script* file waaarmee de compilatie tijd voor beide hoofdprogramma's handig gemeten kan worden.

Test de compilatietijden van de hoofdprogramma's voor verschillende waarden van `N` (b.v. 10, 100, 1000) en `M` (b.v. 10, 100, 1000).

5. Geautomatiseerd Geheugenbeheer

Context

In de slides wordt beschreven hoe de gebruiker verlost kan worden van de vervelende en foutgevoelige taak van het opruimen van objecten (*Geautomatiseerd Geheugenbeheer*).

Opdracht

Neem een niet-triviale applicatie (bijvoorbeeld de "dierentuin" van de module C++) en automatiseer daarvan het geheugenbeheer met behulp van zelfgeschreven "*smart pointer handles*" en "*reference counting*".

Voetnoot

Tegenwoordig (C++11) zitten dit soort "*smart-pointers*" al in de STL.

6. Generieke Algoritmen – *istream* en *ostream iterators*

Opdracht

Schrijf een programma dat een willekeurige reeks gehele getallen van standaard input leest met behulp van een *istream-iterator*.

Schrijf de oneven getallen naar een bestand met behulp van een *ostream-iterator*. Deze getallen worden van elkaar gescheiden door een spatie.
Schrijf de even getallen naar een ander bestand met behulp van een *ostream-iterator*. Deze getallen komen ieder op een aparte regel.

7. Design Patterns – Composite

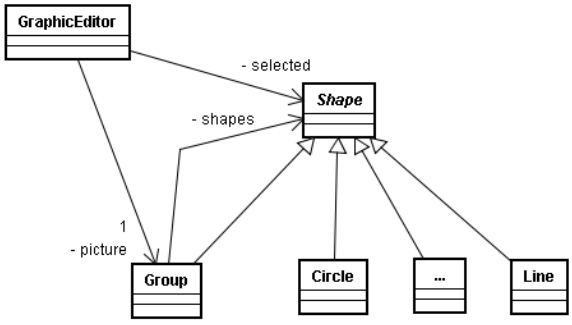
Context

Met behulp van een grafische editor kunnen plaatjes gemaakt worden die cirkels, lijnen, vierkanten en dergelijke bevatten.

Figuren in een plaatje kunnen gegroepeerd worden tot een ***groep*** die weer als één figuur beschouwd kan worden met betrekking tot bijvoorbeeld translatie, kopiëren, verwijderen enzovoort.

Voor het gemak nemen we aan dat bij onze editor precies één plaatje hoort.

Het classe diagram hiernaast geeft de situatie weer. Afbeelden naar C++ levert de volgende classes op (zie bijlage: **`graphics.h`**)



Bij elk `Shape` object hoort een middelpunt dat vastgelegd wordt in een `x` en `y` coordinaat. Zo is het middelpunt van een `Line` object het midden van het lijnstuk en wordt het

middelpunt van een groep bepaald door het gemiddelde te nemen van de x- en y-coordinaten van de middelpunten van de `Shape` objecten waaruit de groep bestaat.

In datamember `angle` wordt de hoek (in radialen) bijgehouden die een lijn maakt met de positieve x-as.

Translatie betekent het verhogen van de middelpunt-coordinaten met de opgegeven waarden.

Voor de elementaire figuren `Circle`, `Line` etc. mogen de `draw()` functies bekend worden verondersteld en hoeven dus niet te worden geïmplementeerd.

Na lezen van bovenstaande moge duidelijk zijn dat een plaatje, in de voorbeeldcode `picture` genoemd, bestaat uit een lijst van figuren (`Shape` objecten) en dat een figuur (`Shape` object) zelf ook weer kan bestaan uit een lijst van figuren (`Shape` objecten).

De editor kent een commando `select(int x, int y)` dat een figuur waarvan het middelpunt overeenkomt met deze (x, y), in de lijst van geselecteerde figuren zet. De editor kent ook het commando `deselect(int x, int y)` dat een geselecteerde figuur met middelpunt (x, y) weer deselecteert.

De editor kent een commando `group`(), waarmee alle geselecteerde figuren samengevoegd worden tot één groep en een commando `ungroup`(), waarmee de als laatste geselecteerde groep verwijderd wordt en de figuren waaruit de groep bestaat weer als aparte figuren aan het plaatje worden toegevoegd.

Opdrachten:

a) Geef de implementatie van de constructoren van de classes `Shape`, `Circle` en `Group`.
b) Geef de implementatie van de `translate()` memberfuncties van de classes `Shape` en `Group`.
c) Waarom heeft de class `Circle` geen `translate()` memberfunctie?
d) Geef de implementatie van de `select()` memberfunctie van de class `GraphicEditor`.
e) Geef de implementatie van de `group()` memberfunctie van de class `GraphicEditor`.
f) Waarom is van het “friend” mechanisme gebruik gemaakt en op welke manier zou het gebruik ervan vermeden kunnen worden?

8. Design Patterns - Visitor

Context

In een dierentuin wordt de verzorging van de beesten zo veel mogelijk geautomatiseerd. De verzorging van een dier bestaat uit verschillende onderdelen die los van elkaar gegeven worden:

a) drinken
b) eten
c) borstelen
d) luchten
e) enzovoort

Natuurlijk krijgen verschillende soorten dieren een verschillende verzorging.

De verzorging vindt plaats per onderdeel per dier, dus eerst krijgen alle dieren drinken, dan eten enzovoort.

Opdracht.

Implementeer de verzorging van de dieren, gebruikmakend van het *visitor* design pattern.

Losse bijlagen

broncode: **`screen.cc`**
broncode: **`Array.h`**
broncode: **`graphics.h`**