PROJEKTRAPPORT

INTRODUKTION TILL PROGRAMMERING

Utförd av:	Edwin Wallin	Dmp1/Dis1
Datum:	2017-12-16	

Innehåll

1 Introduktion

1.1 Bakgrund

Magiska kvadrater är kvadrater av *n x n* antal rutor av heltal varav varje rad, kolumn samt diagonal ska bilda samma summa. De magiska kvadraterna har sitt ursprung i många olika kulturer och var från början sedda som religiösa symboler men uppskattades också av våra förfäder som tidsfördriv och intressanta matematiska problem att lösa. En av de tidigaste kvadraterna man hittat är från ca 2000f.kr i Kina och kallas *Lo Shu*, de ska senare ha spridits ifrån Kina till Indien och senare vidare till arabvärlden varifrån den senare spreds till Europa(sv.wikipedia.org, 2017).

En av de mest kända kvadraterna finns att hämta ur *Melankolin* där utöver de vanliga reglerna det även går att finna summan, trettiofyra, i de fyra hörncellerna, de fyra mittersta cellerna samt i de innersta cellerna i rad ett och rad fyra, samt kolumn ett och kolumn fyra. Andra intressanta iakttagelser man kan göra ifrån Melankolin-kvadraten är att summan av båda diagonaler är lika med summan av de resterande cellerna. Man kan även se att summan av de två övre kvadraterna är lika med summan av de två undre(sv.wikipedia.org, 2017).

Annat än att vara väldigt intressanta, roliga matematiska problem har magiska kvadrater ingen praktisk användning, de kan dock användas för som en mall för geometri och andra områden där symmetri är viktigt, t.ex. har Walter Trump, baserat på William Walkingtons och Inder Tanejas arbete, konstruerat en linjär area kvadrat där arean av cellerna summerade enligt magiska kvadraters regler ger tjugosju(en.wikipedia.org, 2017).

1.2 Målsättningar

Som någon som programmerat i ca 5 år så är målet att bekanta mig mer med det jag ännu inte jobbat mycket med, som t.ex. mer komplex minneshantering samt C som språk för att förstå rötterna till valen de gjort när de designat andra språk som C++, Python e.t.c. Jag tänker även utföra projektet på ett sätt som gör det enkelt att läsa och förstå hur jag tänkt, samt försöka utveckla på idéerna som projektets specifikationer beskriver. Tanken är att sätta ett högre betyg för kodens struktur och effektivitet men den stora fokusen ligger på att utveckla mig som programmerare, oberoende betygets slutliga nummer. Jag tänker även försöka hålla projektet så fri från OS-baserade funktioner som möjligt så den ska kunna kompileras oberoende operativ-system, dock vet jag ej om jag kommer ha tid att testa detta.

.

2 Analys av projektuppgiften

Efter att ha läst uppgiften fullt ut för att få en bild av projektets storlek ser jag redan att även då det är rätt så få uppgifter att utföra, kräver många av dem flera moment, vilket kommer addera upp till ett jämförelsevis stort program, speciellt om det ska skrivas så generaliserat som möjligt med godtycklig dokumentation. Jag hoppar även över extra uppgifterna för fyra och fem i detta stadiet då jag ser att dessa inte kommer kräva något specifikt utav det tidigare programmet utan kan få sina egna funktioner senare.

Lista på krav efter en första genomläsning:

1. Kontroll om en kvadrat verkligen är en magisk kvadrat
2. Manuell inmatning av en hexadecimal magisk kvadrat(ska kontrolleras).
3. Spara en kvadrat till fil.
4. Läsa en kvadrat från fil(ska kontrolleras).
5. Skriva ut en inläst kvadrat till skärmen.
6. Spela ett litet spel där vissa celler tas bort och användaren ska fylla i dessa.
   Spelet får ha flera skilda lösningar.
7. Allt detta ska vara tillgängligt från en meny

0	d	2	f	30
e	8	7	1	30
b	3	c	4	30
5	6	9	a	30
30	30	30	30	30 30

För att förstå vad en magisk kvadrat är ritar jag upp en liten skiss baserat på förklaringen i specifikationen samt lite sökningar på internet. Varje kolumn, rad samt diagonal ska alltså summeras till trettio för att att räknas som en magisk fyrkant i vårat program.

Jag ser redan här att jag kan generalisera programmet rätt rejält genom att införa en **struct**(en.wikipedia.org, 2017) för matriser då strukturen är snarlik en matris(sv.wikipedia.org, 2017). I mitt fall väljer jag att göra en **struct** av **chars** som jag tolkar som hexadecimala nummer, men en **int** hade funkat lika bra. Personligen känner jag att **chars** tar mindre minne och är lättare att hantera i detta fall. Detta kräver dock att jag som sagt kan tolka **chars** som hexadecimala tal och vice versa.

Jag ser även att nummer sex är ett litet spel, jag känner då att eftersom projektet är relativt litet kan jag säkert hinna implementera ett lite ”roligare” spel så jag inför en **struct** specifikt för de magiska kvadraterna så jag kan ha matris och kvadrat funktionerna separata samt ge spelet och andra delar av programmet lite mer specifik funktionalitet utan att gå ifrån matrisernas generalisering. Bland annat känner jag att ett spelet skulle kunna innehålla ett alternativ för att få ledtrådar från en *rekommenderad* lösning, detta för att göra spelet mindre tradigt.

3 Genomförande

För att ens kunna börja på mina magiska kvadrater ville jag skriva kod för en generell matris, så för detta implementerade jag en **struct** med en pekare till dess element samt två heltals variabler som representerade dess storlek samt några funktioner för allokering av minne, deallokering av minne samt en funktion för att kopiera en matris till en annan men även funktioner för tillgång och ändring av matriser så vi slipper tänka på om vi är innanför matrisen när vi vill komma åt dem eller ändra dess värde. Det viktigaste var att våra **get** och **set**-funktioner(ofta kända som getters och setters) (Mutator method, en.wikipedia.org, 2017) då alltså endast kommer åt minnet vi äger och inte går utanför. För allokering och deallokering av minnet skapade jag även en funktion för att allokera och deallokera tvådimensionella fält som jag sedan kunde använda i matrisens initiering.

För att kunna använda matrisen jag implementerat som heltal skapade jag även funktioner för att tolka **chars** som hexadecimala **ints**, detta är lite av ett problem då talen 0-9 och A-F ligger separat i ASCII-tabellen(www.asciitable.com, 2017). Så för att göra om dem använde jag ett litet trick där vi använder en **chars** heltals värde för att räkna ut vad det är lika med. Eftersom 0 som **char** är samma sak som 48 som heltal kan vi helt enkelt köra denna simpla funktion för att gå från hexadecimal **char** till **int:**

```
if(isdigit(inputChar))
   return inputChar - '0'; // Ex med '1' = 49 i ASCII: 49 – 48 = 1
else
   if(isalpha(hex) && tolower(inputChar) >= 'a' && tolower(inputChar) <= 'f')
      return inputChar - 'a' + 10; // Ex med 'b' = 98 i ASCII: 98 – 97 + 10 = 11
```

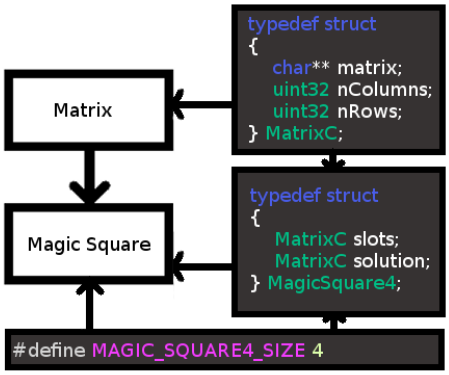
För att göra samma sak fast från heltal till hexadecimal **char** så kan man rätt lätt göra om funktionen en aning men man använder samma princip.

Ett annat problem jag stötte på som försenade utvecklingen lite var när jag behövde läsa ifrån en fil. I det flesta moderna språk så hanteras minnet, om inte helt automatiskt, så till viss del och när man läser en fil i C behöver du alltså i förväg veta vilken storlek på din buffert du behöver när du ska läsa filen. Du kan också använda en förbestämd storlek och hoppas att det är tillräckligt eller om du är tillräckligt avancerad skulle du kunna läsa filen och öka buffertens storlek under läsningens gång, detta skapar dock rätt mycket overhead. Med tanke på att vi vet att våra filer definitivt kommer vara väldigt små så är det lättast att använda det vanliga sättet och ta reda på storleken innan. Det finns funktioner specifikt per operativ system som kan ge dig storleken på en fil, dock är dessa som sagt beroende på operativ systemet du har och jag bestämde mig därför att använda en gammal, klassisk metod. Vi använder en funktion i C som heter **fseek**(en.cppreference.com, 2017) för att hoppa till slutet, ta reda på längden via **ftell**(en.cppreference.com, 2017) och sedan gå tillbaka till via **rewind**(en.cppreference.com, 2017) för att börja läsningen. För att dessutom säkerställa att filerna läses som **char**, alltså som bytes använder vi ”**rb**” när vi öppnar filen istället för bara ”**r**”. Detta sätt är dock begränsat till filer under 4gb, detta baserat på att **ftell** returnerar en **long int** och alltså inte får plats med större tal, men med tanke på våra specifikationer är detta helt enkelt inget problem. Läs bilaga 1 för en genomgående presentation av koden.

De flesta andra delar av programmet var rätt så lätt att implementera när hjälp-funktionerna var kodade. T.ex. så vid inmatning av den magiska kvadraten manuellt så kunde vi skapa en funktion som använde de redan satta funktionerna **MS4_isValidCharacter** samt **MS4_setSlot** utan att behöva tänka så mycket extra på om man läser rätt minne eller om karaktären är en giltig

hexadecimal karaktär e.t.c. I och med det så satte jag lite extra tid på att lägga till felmeddelanden för när vi kollar om kvadraten är *”riktig”* och annan finslipning.

4 Resultat



Min slutgiltiga struktur för huvudkomponenterna, alltså mina data strukturer blev något i stil med illustrationen till höger. Jag har alltså en datastruktur vid namnet **MatrixC** som innehåller pekaren till minnet där vår matris ligger, samt dess storlek i form av antalet kolumner samt antalet rader. Denna struktur har ett antal hjälpfunktioner som du kan läsa mer om i koden som gör initialisering, minneshantering samt hantering av matrisen lättare. Med hjälp av denna matris kunde jag sedan enkelt implementera funktioner specialiserade för de magiska kvadraterna då jag kände för att hålla dessa isär, så jag gjorde en egen datatyp speciellt för kvadraterna kallad **MagicSquare4.** Fyran i **MagicSquare4** existerar endast för att den enda typen av magiska kvadrater vi behöver stödja är av storlek fyra och om vi någonsin skulle utveckla detta program vidare vill jag att det ska vara tydligt direkt ifrån gränssnittet att det rör sig om just *4 x 4* kvadrater för de redan implementerade funktionerna.

Det slutgiltiga gränssnittet jag bestämde mig för blev kanske inte riktigt optimal då jag senare insåg att i varje **MagicSquare4** integrera en matris för den rekommenderade lösningen inte var direkt användbart annat än i spelet, så detta skapade endast mer overhead, extra funktioner samt tog upp mer minne än som hade behövts . Dock så med det nuvarande gränssnittet är det väldigt tydligt vad som görs av varje funktion, speciellt då i spelet. Hade jag gjort om det hade jag definitivt hållit mig till en matris per magisk kvadrat.

Det slutgiltiga gränssnittet för de magiska kvadraterna illustreras nedan. Vill du ha en mer ingående förståelse för hur funktionerna fungerar rekommenderar jag att läsa igenom koden.

MagicSquare4 gränssnitt

```
MS4_init(); // Kallas först i programmet

// Skapa kvadrat
MagicSquare4 sq;

// Allokera minne
MS4_initSquare(&sq);

// Exempel
char s = MS4_readSlot(&sq);

MS4_setSlot(&sq, column, row, character);

// Ge tillbaka minnet
MS4_destroySquare(&sq);

MS4_quit(); // Kallas sist i programmet
```

MatrixC gränssnitt

```
// Använder matrisens gränssnitt: allokerar minne och sätter storlek
initMatrixC(&sq->slots, MAGIC_SQUARE4_SIZE, MAGIC_SQUARE4_SIZE);
initMatrixC(&sq->slots, MAGIC_SQUARE4_SIZE, MAGIC_SQUARE4_SIZE);
...
// Använder matrisens gränssnitt: hämtar element från matris
return getElementMatrixC(&sq->slots, column, row);
...
// Använder matrisens gränssnitt: ändrar på en matris
if(MS4_isValidCharacter(character))
{
   setElementMatrixC(&sq->slots, column, row, character);
}
...
// Använder matrisens gränssnitt: frigör minnet
destroyMatrixC(&sq->slots);
destroyMatrixC(&sq->solution);
```

Tack vare detta sätt att strukturer gränssnittet behövde jag endast programmera saker så som **get/setElement** mer eller mindre en gång och minska antalet kodrader som annars hade behövts dupliceras över programmets alla rader. Till exempel behöver jag bara kolla om de efterfrågade *”kordinaterna”* är inom matrisens storlek en gång då det sköts av matrisen. Detta gör det inte bara lättare att utveckla programmet nu, men ska det läsas av andra eller av mig själv senare i livet kommer det vara mycket mer lättförståeligt vad som händer.

De olika specifikationerna kan med hjälp av detta gränssnitt rätt smärtfritt implementeras:

4.1 Kontroll om det är en magisk kvadrat

Detta kan nu enkelt kollas med hjälp av våran hjälpfunktionerna för våran matris. Vi kan enkelt kolla summan genom att **loopa**(en.wikipedia.org, for/while loop**,** 2017) igenom varje kolumn samt rad och hämta varje element via **getElementMatrixC** så slipper vi tänka på att säkerställa in-parametrarna eftersom det redan hanteras av funktionen. För att underlätta denna process någorlunda skapade jag även här lite hjälpfunktioner för att summera diagonalerna, kolumnerna och raderna.

4.2 Manuell inmatning

Detta var också rätt så trivialt att implementera när hjälpfunktioner var satta. En enkel dubbel **for-loop**(en.wikipedia.org, 2017) som frågar användaren om vad hen vill lägga i en plats. Även här behövdes ingen speciell extra kod då det mesta hanteras av matrisernas hjälpfunktioner.

4.3 Spara till fil

Detta tog lite mer än att bara kalla ett par hjälpfunktioner. Men jämfört med att läsa en fil är det rätt smärtfritt tack vare C-språkets funktion **fputs**(en.cppreference.com, 2017). För att göra programmet lite mer användarvänligt implementerade jag även en metod för att kolla om en fil redan existerar så användaren själv kunde välja om de ville skriva över filen om den redan existerade. Den sista byggstenen här var att konvertera **MagicSquare4** till en text sträng, därav kom **MS4_toString** att existera.

4.4 Läsa från fil

Som tidigare nämnt var detta lite av en utmaning, jag har redan förklarat varför i kapitel 3 men kan även här nämna det. När man läser en fil behöver du ha en buffert där du kan spara all data du får ifrån filen. Vet du dock inte hur stor filen är vet du inte heller hur stor buffert du bör allokera. Det finns förmodligen en hel drös lösningar för detta men de tre huvudsakliga idéerna för att lösa detta är att:

1. Allokera en större buffert än du räknar med behöva och hoppas att det är tillräckligt
2. Konstant ändra storleken dynamisk baserat på hur mycket data du läser in
3. Fråga operativsystemet efter filen storlek och allokera en buffert baserat på det

Nummer 1 är helt enkelt för extremt specifika fall där du vet exakt hur stora filer du har, vilket är extremt ovanligt och för att göra programmet lite mer generaliserat är nummer 2 mer användbart, dock är att konstant allokera(sv.wikipedia.org, 2017), kopiera och deallokera minne väldigt långsamt och skapar overhead som vi helst vill undvika så jag valde nummer 3. För att ta reda på storleken finns en hel drös metoder, många är dock helt baserade på vilket operativ system du kör. Så för att slippa detta använda jag vanliga C funktioner för att få fram storleken. Läs bilaga 2 för mer information om hur man hittar en fils information. Efter detta är det dock rätt lätt då vi kan köra **fread**(en.cppreference.com, 2017) för att läsa datan direkt in i buffert, så lägger vi dessutom på en ’\0’(www.asciitable.com, 2017) på slutet för att göra text strängen säker för användning.

4.5 Skriv till skärm

Skrivning till skärmen är ett rätt enkelt problem som dock också kan lösas på oändligt många sätt. Det enda jag egentligen behövde göra är att skriva två kapslade **for-loopar** och sedan kalla min någon av mina hjälpfunktioner, **MS4_readSlot** eller **MS4_readSol** beroende på vilken komponent av kvadraten vi vill ha som utskrift.

4.6 Spela ett litet spel

För att få en bild av hur spelet skulle fungera citerar jag specifikationen:

”Det skall gå att spela ett litet ”fylla i” spel. Från en giltig hex-kvadrat skall ett slumpmässigt urval av siffrorna tas bort (max 8 siffror). Det är nu användarens uppgift att fylla i de luckor som uppstått så att kraven för hex-kvadraten är uppfyllda. Det får finnas flera lösningar.”

Den sista meningen indikerar på att vi inte behöver behålla den riktiga lösningen och alltså krävs det inte att man har den struktur jag valde för **MagicSquare4** dock ville jag fortfarande behålla detta då jag kände för att utveckla spelet till ett ändå på något sätt roligt spel och gärna ville implementera en funktion för att ge tips till spelaren för att undvika att det blev alldeles för tradigt.

För att underlätta spelets utveckling bröt jag isär problemen och fann att ett antal funktioner jag ännu inte implementerat behövdes, speciellt en viktig komponent var **randomInt** som slumpar fram ett tal mellan **min** och **max.** Som med det mesta i min kod kunde de varit tillräckligt att slumpa fram tal mellan 0 och **max** och simplifiera funktionen, dock kände jag att jag lär mig mer på att generalisera så mycket som möjligt och göra funktionen lite mer användbar. Med **randomInt** implementerat var det bara att skapa lite små funktioner som gjorde utskrifter till skärmen samt frågade användaren för lite inmatning och köra vår check för giltiga kvadrater för att se att om den var löst eller inte(**rand**, en.cppreference.com, 2017).

4.7 Meny

Hade jag haft tiden hade jag gärna spenderat timtals på att skapa ett bibliotek för riktigt häftiga menyer och grafiska interfaces för konsolen men tyvärr hade jag inte den tiden och därför är mina menyer simpla **switchar**(switch, en.cppreference.com, 2017) som jag kör ett nummer igenom.

5 Diskussion

Detta var ett rätt trivialt projekt som jag definitivt kunde gjort bättre men som jag är rätt glad att jag fått göra även om jag ofta kände att de jag gjorde var lite för enkelt eller lite för tråkigt. Det finns mycket i C som ger en god förståelse för hur andra språk fungerar och varför vi behöver göra vissa saker på ett sätt och inte ett annat. En god förståelse för C ger en bättre förståelse för i princip vartenda annat programmeringsspråk som finns. Jag har tidigare programmerat i ca 5 år med, för det mesta, programmeringsspråket C++ som är en utveckling av C, men aldrig riktigt dykt ner i grunderna och alltså inte haft riktigt den förståelsen för riktig minnes hantering och annat som jag egentligen skulle vilja ha.

Detta projekt har också get mig mer respekt för planering och klok utvärdering då jag nu inser att jag sköt upp projektet lite för länge och både underskattar och överskattar vissa delar av min kunskap. Även om jag påstår att mycket var för enkelt finns det mycket jag kunde gjort bättre och är som sagt glad att få göra denna utvärdering även om det är rätt dystert att sitta och skriva.

6 Referenser

1. Magisk Kvadrat, https://sv.wikipedia.org/wiki/Magisk_kvadrat, 2017
2. Magisk Kvadrat, https://en.wikipedia.org/wiki/Magic_square, 2017
3. For loop, https://en.wikipedia.org/wiki/For_loop, 2017
4. While loop, https://en.wikipedia.org/wiki/While_loop, 2017
5. Struct i C, https://en.wikipedia.org/wiki/Struct_(C_programming_language), 2017
6. Matris, https://sv.wikipedia.org/wiki/Matris, 2017
7. Get/Set metoder/funktioner, https://en.wikipedia.org/wiki/Mutator_method, 2017
8. ASCII teckentabell, http://www.asciitable.com/, 2017
9. fseek, http://en.cppreference.com/w/c/io/fseek, 2017
10. ftell, http://en.cppreference.com/w/c/io/ftell, 2017
11. rewind, http://en.cppreference.com/w/c/io/rewind, 2017
12. fputs, http://en.cppreference.com/w/c/io/fputs, 2017
13. fread, http://en.cppreference.com/w/c/io/fread, 2017
14. Minnes allokering, https://sv.wikipedia.org/wiki/Minnesallokering, 2017
15. rand, http://en.cppreference.com/w/c/numeric/random/rand, 2017
16. switch, http://en.cppreference.com/w/c/language/switch, 2017

7 Bilagor

Bilaga 1: Läsning av fil till sträng, PDF
Bilaga 2: Fil information, PDF