Grundläggande programmering med C++, kurskod IT120G. Sommar 2017 (ST2017)

Inlämningsuppgift 1: Roulett

Joel Ottosson, 2017-07-14, 19960404-5170

Innehåll

[Introduktion 1](#_Toc487802636)

[Problembeskrivning 1](#_Toc487802637)

[Givna problem 1](#_Toc487802638)

[Dolda problem 1](#_Toc487802639)

[Antagande och Krav 2](#_Toc487802640)

[Lösningsdesign 3](#_Toc487802641)

[Allmänt tillvägagångssätt 3](#_Toc487802642)

[Lösning på problemen 4](#_Toc487802643)

[Diskussion 8](#_Toc487802644)

# Introduktion

Denna rapport är en beskrivning av hur programmet ”main” skrevs. Programmet syfte är att imitera kasinospelet roulett på ett så realistiskt sätt som möjligt. Nedan följer en beskrivning av de krav som ställdes på detta spel, vilka del problem som uppenbarades under skrivandets gång, egen satta krav och antagande. Detta följt av en beskrivning av processen som ledde fram till det färdiga programmet. Sedan ges lösningar på de identifierade problemen och rapporten avslutas med en diskussion kring kvalitén på programmet och eventuella förbättringar.

# Problembeskrivning

## Givna problem

Den givna uppgiften var att skapa ett virtuellt roulettspel genom kodning i programmeringsspråket C++. Spelet skulle ha strukturen:

1. Användaren väljer insats.
2. Väljer färg eller nummer.
3. Hjulet snurrar och en siffra slumpas fram.

**Användaren väljer insats.** Spelaren väljer en insats på 100, 300 eller 500.

**Väljer färg och nummer**. Om spelaren vinner på färg ska 2 gånger insatsen erhållas och om spelaren vinner på nummer ska 10 gånger insatsen erhållas.

**Hjulet snurrar och en siffra slumpas fram.** En siffra mellan 1 och 36 ska slumpas fram där jämnt nummer är svart och udda nummer är rött. Resultatet ska sedan redovisas för spelaren där vinsten eller förlusten ska visas tillsammans med den totala vinsten. Därefter ska spelaren få ett val om hen vill spela igen eller inte, detta om pengar finns för att kunna spela igen.

Ett sista krav som var uttryckligen givet i instruktionerna är att spelaren, när hen först öppnar programmet, ska börja med en mäng pengar av 1000.

## Dolda problem

**Ska färgerna ”Röd” och ”Svart” representeras med heltal eller boolean?**

Olika koncept måste representeras i programmet. Konceptet ”att gissa på en färg” lika så och frågan är då vilken datatyp fångar bäst denna gissnings beteende?

**Problem: Återkommande problem av inmatning följt av en tilldelning av variabel.**

Det är ett antal gånger som spelaren ska mata in ett värde och sedan ska en variabel tilldelas ett värde beroende på denna inmatning. En lösning som fungerar till denna problem-familj vore att föredra.

**Problem: Tudelningen**

Beroende på om spelaren väljer att satsa på färg eller heltal tar programmet 1 av 2 vägar. Hur ska detta lösas och struktureras?

**Problem: Nested switch case**

En switch-case-sats jämför en variabel med ett antal alternativ. Detta kan störas om denna variabel ändrar värde under tiden värdena jämförs. Detta är en risk med att använda sig av en nested-switch-case-sats som beror på samma variabel.

# Antagande och Krav

**Krav: Programmet ska inte krascha vid inmatningen på grund av att en datatyp försöker tilldelas en annan.**

Detta är ett problem som inte bör finnas i en kod. Om användaren bes mata in en siffra och användaren råkar skriva ”2a” så ska programmet inte krascha.

**Krav: Programmet ska inte krascha vid inmatningen på grund av att heltalet är större/mindre än förväntat.**

Ett bra program bör inte krascha om användaren matar in ”3” när det maximala förväntade beloppet var 4.

**Krav: Spelaren får inte satsa mer än vad hen har**

Om uppgiften relateras till verkligheten så bör inte en spelare kunna satsa mer pengar än vad hen kan betala. Att tillåta detta vore dålig business.

**Krav: Stanna programmet**

För att öka spänningen när spelaren spelar så ska programmet vänta några sekunder efter det att hjulet snurrats tills det att resultatet ges.

* Meningen ”Lyckas användaren vinna ett spel ska programmet skriva ut aktuell vinst samt den totala vinstsumman (om användaren spelat flera gånger).” som hittas i de givna kraven kan tolkas lite olika. ”totala vinstsumman” kan antingen tolkas som den totala mängden pengar som spelaren har kvar till sitt befogande eller den mängd pengar som faktiskt har vunnits. I detta program antas den första tolkningen.
* I detta program antas det att spelaren endast ska bli erbjuden att spela spelet igen om hen har positivt med pengar.
* Spelaren ska kunna skriva in hur många ogiltiga inmatningar som helst vid varje inmatningstillfälle.
* Spelet ska vänta ett tag mellan det att hjulet börjar snurra tills det att resultatet ges. Allt för spänningen.

# Lösningsdesign

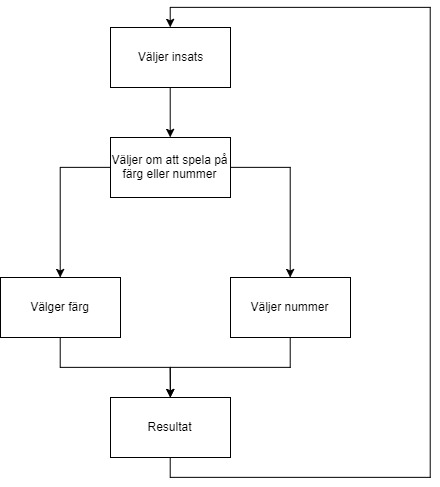
### Allmänt tillvägagångssätt

Mycket av uppgiften har bara lösts via ”trial and error”. Utöver det har det varit viktigt att:

1. Verifiera att det alltid funkar så här långt…
2. Börja med enkla flödesdiagram och jobba neråt.

Med den första punkten menas mer eller mindre att inte stora delar kod ska skrivas åt gången utan istället skrivs ett par rader på sin höjd och sedan kontrolleras om dessa rader gör vad som önskas. Förutsägelser gör dessutom för att säkerhetsställa att programmet är förstått och gör som menat.

Den andra punkten handlar om att dela upp problemet i mindre delmål och dessa i ännu mindre. Det enklaste flödesschemat såg ut som med bilden nedan.



Efter detta schema är gjort kan varje box ses som sitt eget problem och samma sak kan göras därifrån.

Då inmatning sker ett antal gånger var det ett viktigt problem att lösa på ett standardiserat sätt. Samma sak med behandlingen av detta indata. Medan processen fortgick uppstod dolda problem som redogjorts ovan och lösningar visas nedan. Ett exempel på ett sådant är: Spelaren ska inte kunna satsa mer än vad hen har.

Annars löstes uppgiften i kronologisk ordning. När programmet tycktes färdig genomfördes en rad tester av t.ex. olika inmatningar för att säkerhetsställa att programmet betedde sig som väntat. Det mesta fungerade som det skulle men här korrigerades även ett antal processer. Efter detta tittades programmet igenom för att se om det användes några onödiga variabler eller liknande. Därefter letades det efter godtyckliga sifferkonstanter som skulle passa bättre som variabler eller konstanter. Tillslut kontrollerades kommentarerna till programmet och indenteringen justerades.

### Lösning på problemen och krav

**Ska färgerna ”Röd” och ”Svart” representeras med heltal eller boolean?**

Detta handlar egentligen om smak. Det går att lösa problemet oavsett vad som väljs. Det mest naturliga alternativet, i och med att Röd och Svart är binärt, är boolean. Fördelar med detta alternativ är dels att datatypen boolean tar mindre minne än Integer. Vidare är en fördel att om det inte är sant (rött) så är det falskt (svart). Det går alltså att beskriva t.ex. svart som: ! RÖT, något som skulle kunna leda till elegantare kod. Nackdelarna som uppstår är att det blir två fundamentalt skilda gissningar som kan göras. Antingen gissas det på ett heltal eller på ett booliskt värde. Det gör att den befintliga *guess* variabeln enbart representerar siffergissningar och en ny variabel som representerar färggissningar måste läggas till. Detta kan medföra att det blir förvirrande då det är fler variabler med i bilden som för spelaren representerar samma sak; programmet blir mer distanserat från verkligheten. Argumentet för boolean skulle kunna göras av att det blir mer läsbart i koden om *false* och *true* skrivs istället för t.ex. *-223* och *224,* dock spelar det inte någon roll om istället konstanter sätts till dessa värden. Det kan tänkas att det är svårt att skilja på färg- respektive heltalsgissningar om båda representeras av en heltalsvariabel. Men om färg konstanterna sätts till negativa heltal så är det enkelt att särskilja på de 2 olika gissningarna. Dessutom gör detta val att om antalet heltalsgissningar skulle öka, från 36 till t.ex. 54, skulle det inte innebära att en färg och en siffra skulle representeras av samma heltal.

**Krav: Programmet ska inte krascha vid inmatningen på grund av att en datatyp försöker tilldelas en annan.**

Detta problem löses genom att inmatningen sätts i en sträng-variabel och kan därför inte krascha. Sedan skickas denna sträng till funktionen *inputHandler*. Denna funktion använder sig av en *try-catch­*-sats som testar om programmet kraschar av att försöka sätta in värdet i strängen i funktionen *stoi* (string to integer). Om programmet kraschar bes spelaren att mata in ett nytt värde. Detta fortsätter tills det att programmet inte kraschar och heltalet som matats in returneras. Loop fenomenet löses via en *while-loop*. Anledningen till att *inputHandler* lades som en egen funktion var att just den processen återkommer ett antal gånger.

**Krav: Programmet ska inte krascha vid inmatningen på grund av att heltalet är större/mindre än förväntat.**

Detta problem löses med en *while-loop* och en if sats. När *inputHandler* returnerar ett heltal tittar programmet om detta heltal är innanför spannet ifråga. Är den det utförs kommandot *break* som gör att programmet hoppar ur loopen. Om det inte är innanför spannet så får spelaren ett fel-meddelandet och är ombedd att mata in ett nytt värde.

Vid utformningen av denna lösning var en *do-while*-sats under bestänkning, men då felmeddelandet endast ska köras om spelaren matat in fel så sorterades det alternativet bort.

**Krav: Spelaren får inte satsa mer än vad hen har**

Detta problem löses med en *do-while-*sats som efter alla andra tester kontrollerar att tillräckligt med pengar finns för att spelaren ska få fortsätta.

**Problem: Återkommande problem av inmatning följt av en tilldelning av variabel.**

Det är ett antal gånger som spelaren ska mata in ett värde och sedan ska en variabel tilldelas ett värde beroende på denna inmatning. Detta problem är löses varje gång med 2 steg.

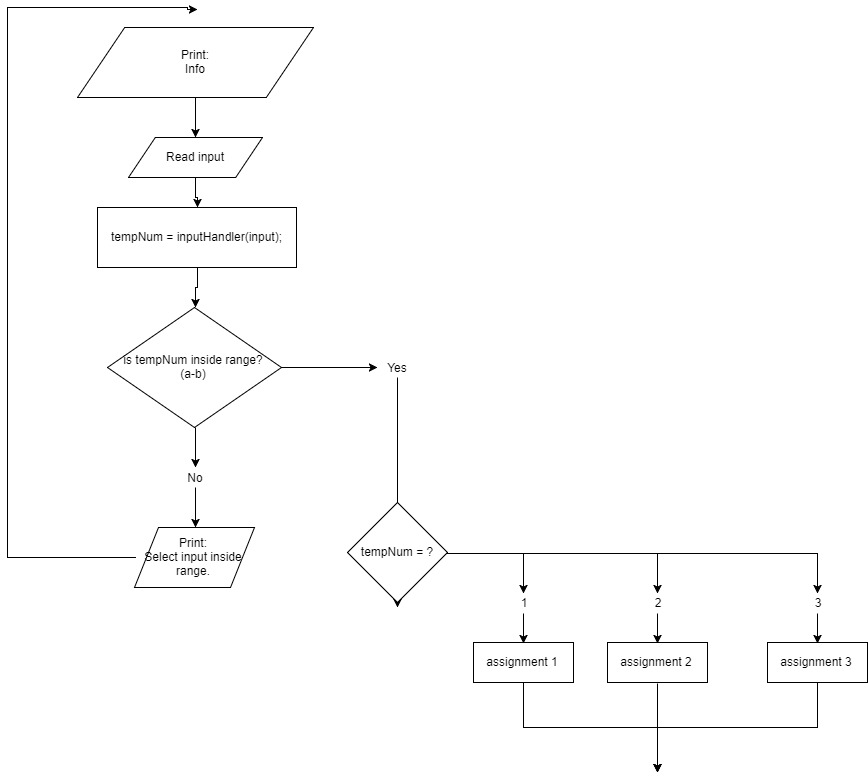
1. En kod som upprepas till dess att spelaren har matat in ett lämpligt värde.

Denna del löses med en *while-loop* som varje gång den körs visar spelaren sina alternativ följt av en begäran av inmatning. Inmatningen skickas sedan till funktionen *inputhandler* som i sin tur kan begära nya inmatningar om det är så att spelaren inte har matat in ett heltal. Inmatningen kontrolleras sedan med en *if-sats* om den är giltig och om sant utförs kommandot *break* som får programmet att hoppa ut ur loopen. Om falskt visas ett felmeddelande och loopen börjar om.

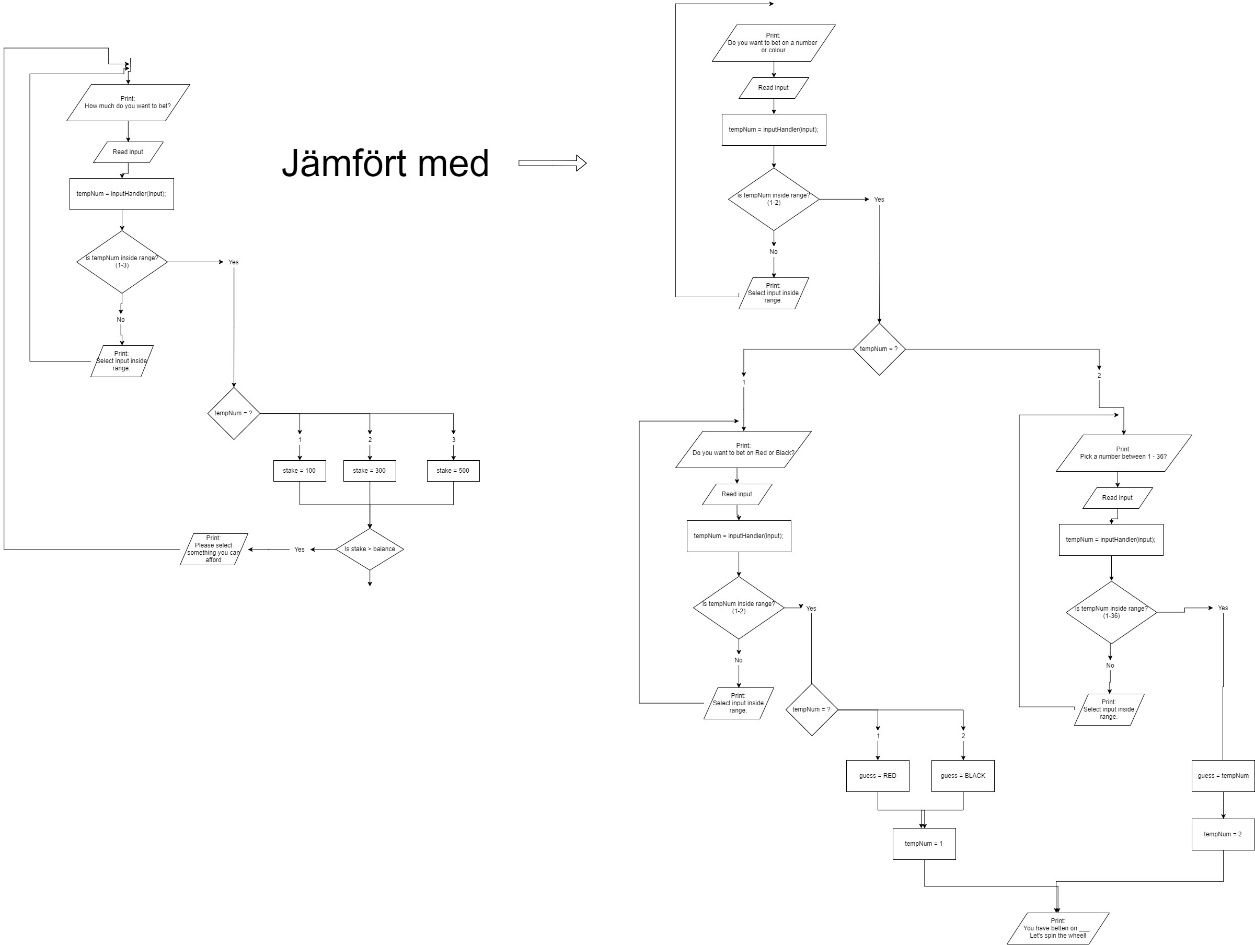
1. Tilldelning av variabel

Detta görs genom en ­*switch-case-sats* som går igenom ett alternativ i taget och utför koden som är associerade med inmatningen.

Se standardlösningen i flödesschemaform nedan.



**Problem: Tudelningen**

Normalt när spelaren får välja en väg i spelet resulterar det i 1 kommando istället för 1 annat, stake = 300 istället för 500, men då spelaren väljer mellan att satsa på nummer eller färg är det flera rader kod som antingen ska köras eller hoppas över. 

Detta vägskäl skulle kunna lösas med en ifsats högst upp som drar en linje i mitten av flödesschmat ovan följt av den standardiserade lösningen 2 gånger. Ett annat alternativ vore att införa en variabel, *bool guessOnNumber*, som på standardiserat sätt tilldelas sant om spelaren satsar på ett nummer och falskt om spelaren satsar på färg. Efter det går det att ha 2 separata if-satser där rätt kod utförs. Det senare alternativet skulle troligen leda till mindre läsbarhet och större komplexitet då fler variabler lägs in i bilden. Den första lösningen är på så sätt bättre men bryter också den standardiserad lösnings metoden. Därför är lösningen i programmet en standardiserad lösning i den standardiserade lösningen. Alltså spelaren väljer att satsa på färg eller nummer vilket leder dem till en rad i en switch-case-sats. Här begärs spelaren åter att mata in data vilket behandlas på standardiserat sätt. Detta gör att lösningen blir en nested-standardlösning.

**Problem: Nested switch case**

Switch-case-satsen går igenom ett antal alternativ och tittar om en variabel är lika med det alternativet. Det kan då uppstå problem om man under tiden ändrar på denna variabel. Detta är något som görs i detta program. Anledningen är för att slipa införa fler variabler som inte tillför läsbarhet eller allmän förståelse. Problem stoppas ändå från att uppkomma då i slutet av varje block där variabeln ifråga ändras så återställs den till sitt ursprungliga värde.

**Krav: Stanna programmet**

Ett eget krav var att programmet skulle stanna i ett antal sekunder då hjulet snurrade innan resultatet gavs ut. Detta löstes med Sleep-funktionen som kan hittas i windows.h biblioteket.

# Diskussion

**Generiskt**

Programmet är relativt generiskt skulle jag hävda. Den använder sig av variabler och konstanter som ett casino skulle kunna vilja ändra på:

* Startvärdet
* Faktor vid vinst i både gissning på nummer och färg.

En konstant som skulle kunna tänkas läggas till vore antalet siffror på rouletthjulet. Detta gjordes inte dels på grund av att en sådan sak normalt inte ändras. Sedan är det en ytterligare variabel som gör programmet mer komplext och dessutom skulle denna variabel heta något i stilen med *maxNum* vilket jag ser som en försämring i läsbarhet. En process som skulle kunna göras generisk är hur mycket spelaren vill satsa. Istället för att ge 3 alternativ skulle spelaren kunna skriva in själv hur mycket hen vill satsa, förutsatt att hen har tillräckligt. Detta gjordes dock inte då det står uttryckligen i de givna kraven att det nuvarande sättet ska utföras.

Som nämnt är inmatning och tilldelning av variabel en vanlig process i detta program och har därför tilldelats en standardiserad lösning. Här kan det vara mycket lockande att skapa en funktion för denna standard istället för att kopiera den med mindre förändringar. Anledningen till att det inte gjort är dels att min programmeringsförmåga inte är tillräcklig, det är små skillnader mellan olika gånger standarden används, något jag inte vet hur jag ska lösa. T.ex. så är det olika information som ska ges användaren, det är olika spann som är tillåtna och det är olika variabler i fråga. Möjligen är det även så att på denna storlek av program skulle en funktions lösning öka komplexiteten och minska lösningen vilket gör att även om jag kunde göra det så skulle det vara till ingen nytta.

**Läsbarhet**

Här tycker jag programmet lyckades bra. Om man skriver ut huvudkommentarerna, de som står på egen rad lång till vänster, så får man:

// Welcome to the casino

// Game loop starts

// Handling of input to stake

//Handling input to decide if player will bet on colour or number

// Guess is assigned a value based on the input. guess =(1, ..., 36, RED,

// Tell the player what they betted on

//Create the winning number and the colour of which

// Check if the player won

// Adjust the value of balance and gameWon

//Give the player the results

//Check if player has positive balance and wants/can play again

// Handling input to gameOver

// gameOver is assigned a value based on the

//Casino saying goodbye

En person som läser detta utanför sitt sammanhang skulle nog kunna få en ganska okej uppfattning om vad som händer i programmet och som stöd till koden är det nog en jätte hjälp.

Variablerna har lättförståeliga namn som relativt bra beskriver vad de gör. Undantagen är väll tempNum som är en alltiallo variabel som används när heltal behöver transporteras eller manipuleras.

Något som förminskar läsbarheten ganska mycket är *condition ternairy operand*. T.ex.

(winningColour == BLACK) ? "BLACK " : "RED "

Den gör koden mer elegant och effektiv men kan vara tankemässigt jobbigt att titta tillbaka på. Det var svårt att avgöra om det positiva övervägde det negativa här men programmet innehåller i alla fall ett antal sådana.

**Korrekt inmatning**

En fråga som kan vara svår att svara på är ”Vad är en korrekt inmatning?”. Nedan ges en lista på olika typer av inmatningar och hur programmet reagerar på dessa.

|  |  |  |
| --- | --- | --- |
| Inmatnings typ | Ex | Händer i programmet |
| String | asaa | Programmet säjer ifrån i inputHandler funktionen och ber om en ny inmatning. |
| Integer (Utanför range) | 1222 | Programmet når inte break kommandot för att ta sig ut ur loopen utan ber om ett nytt värde. |
| Integer String | 3sad | Programmet klagar inte utan sträng delen kastas bara. |
| String Integer | Ss1 | Programmet reagerar som om det endast vore en sträng som inmatats och heltalsinformationen förloras. |
| Integer (space) String | 1 aaa | Här klagar inte programmet vid första inmatningen utan läser in siffran. Men nästa gång en inmatning är väntad så tar den in bokstäverna som skrivits in och ger ett felmeddelande. |
| Integer (space) Integer | 1 32 | Här klagar inte programmet någon gång men det andra heltalet som skrivit kommer bli inmatningen till nästa fråga. |
| String space) Integer | Ass 11 | Här kommer först ett felmeddelande upp men då det följs av en ny inmatningsbegäran så kommer heltalet att läsas in denna gång. |

**Onödiga heltalskonstanter**

De flesta heltalskonstanter som är inskrivna i programmet är de som beskriver spannet som en inmatning är tillåten att ha. Detta spann skiftar från olika delar i programmet och att införa dessa som variabler eller konstanter skulle minska läsbarheten och knappt öka flexibiliteten på något sätt. De representerar inte heller något verkligt som är viktigt för användaren utan endast antalet möjlig val vid en inmatning. Det span som möjligen skulle kunna tänkas göras till en konstant är spannet av heltalsgissningar.