Norges teknisk–naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

TDT4102 Prosedyre- og objektorientert programmering Vår 2017

Øving 12

Frist: 2017-04-25

Mål for denne øvingen:

- Unntakshåndtering og egne unntaksklasser
- Bruke det vi har lært i emnet

Generelle krav:

- Bruk de eksakte navn og spesifikasjoner som er gitt i oppgaven.
- Det er valgfritt om du vil bruke en IDE (Visual Studio, XCode), men koden må være enkel å lese, kompilere og kjøre.
- Skriv den nødvendige koden for å demonstrere implementasjonene dine

Anbefalt lesestoff:

- Kapittel 17 & 18, Absolute C++ (Walter Savitch)
- Artikkelen Mathematics of Sudoku på Wikipedia

MERK: Denne øvingen har totalt 130%. Øvingen består av to deler — *SafeArray* og *Sudoku* — hver med flere oppgaver. Du trenger **70%** korrekt for å få øvingen godkjent, og kan selv bestemme hvilke oppgaver du gjør.

Det forventes at du kan demonstrere at koden du lager kjører korrekt.

Del 1: SafeArray

1 SafeArray (20%)

Tidligere i øvingsopplegget har vi lært at tabeller (arrays) i C++ har enkelte begrensninger i forhold til det vi kanskje er vant med fra andre programmeringsspråk, i og med at vi må passe på at vi ikke kan endre størrelsen på dem enkelt, og heller ikke enkelt slå dem sammen. I denne oppgava skal vi derfor lage oss en klasse som oppfører seg litt mer som Python-lister, og se at vi kan få akkurat den funksjonaliteten vi vil, ved å bygge sammen konsepter vi har lært tidligere i øvingsopplegget.

Først noen generelle krav til klassen `SafeArray`:

1. `SafeArray` skal være en template-klasse slik at den kan holde alle datatyper som kan default-konstrueres.
2. Alle medlemsvariabler skal være private.
3. Det skal være mulig å lagre elementer i klassen.
4. Dataene skal lagres i en dynamisk tabell internt i klassen.
5. Det skal være mulig å *finne ut* hvor mange elementer det er plass til.
6. Det skal være mulig å *endre* hvor mange elementer det er plass til.
7. Dersom man reduserer størrelsen skal de siste elementene som det ikke er plass til, gå tapt.
8. Dersom man øker størrelsen, skal alle nye elementer være default-initialisert, mens alle gamle skal bevares.
9. Det skal være mulig å bruke []-operatoren til å lese ut, og skrive til elementer.
10. `SafeArray` skal ha konstruktører, slik at man kan opprette ei liste med N elementer, som alle er default-initialisert.
11. `SafeArray` skal ha en destruktør som rydder opp i allokert minne.
12. `SafeArray` skal ha kopikonstruktør og operator= som implementerer deep-copy, slik at vi trygt kan kopiere SafeArrays.
13. `SafeArray` skal kaste exceptions hvis man forsøker å utføre ulovlige handlinger (Dette vil bli spesifisert nøyere lengre ned).

Du står rimelig fritt til å velge hva du kaller medlemsvariabler, og medlemsfunksjoner, men **det forventes at samtlige krav testes[1] og demonstreres i `main()`**. Under får du allikevel en kort oppskrift på rekkefølgen det kan være nyttig å angripe problemet i:

a) Opprett template-klassen `SafeArray` (Krav 1)

b) Legg til de medlemsvariablene du trenger for å oppfylle krav 2, 3 og 4

c) Skriv konstruktørene og destruktørene som trengs for å oppfylle krav 10 og 11

d) Skriv de(n) medlemsfunksjonen(e) som trengs for å oppfylle krav 5

e) Skriv de(n) medlemsfunksjonen(e) som trengs for å oppfylle krav 9

[1] Vær oppmerksom på at funksjoner i templateklasser som ikke brukes, ikke nødvendigvis kompileres, så det er enda viktigere enn tidligere at man tester grundig

f) Skriv konstruktør(ene) og medlemsfunksjon(ene) som trengs for å oppfylle krav 12

g) Skriv testkode i `main()` som oppretter et `SafeArray`, fyller det med verdier, og skriver disse ut
Forsikre deg også om at du kan endre verdiene etter at de er lagt inn, samt at forskjellige datatyper, som f.eks. `string` og `double` kan lagres i `SafeArray`et.

h) Skriv de(n) medlemsfunksjonen(e) som trengs for å oppfylle krav 6, 7 og 8
Utvid testene dine, slik at du forsikrer deg om at kravene er oppfyllt.
Hint: Du må lage nye tabeller, og kopiere innholdet

i) Utfør følgende test i main()

- Opprett et nytt `SafeArray` med plass til 10 elementer av type `int`.
- La disse elementene være tallene 0-9
- Gang hvert tall med posisjonen sin, slik at tabellen nå inneholder N*N på posisjon N.
- Skriv ut tabellen
- Lag et annet `SafeArray` som kopi av det første, ved hjelp av =.
- Reduser størrelsen på `SafeArray`-et du nettopp lagde til halvparten.
- Legg til 5 til alle tallene som nå er i det nye SafeArrayet.
- Øk størrelsen på det nye `SafeArray`-et til 10 igjen.
- Skriv ut begge `SafeArray`-ene, og bekreft at svaret stemmer med det du forventer.

2 FancySafeArray (20%)

I forrige oppgave la vi grunnarbeidet for å lage en litt mer praktisk tabell-implementasjon enn de enkle dynamiske tabellene som C++ har i grunnlaget. Vi har allerede en tabell som kan lagre fritt valgte datatyper, og endre størrelse ettersom vi trenger det, men hva om vi har lyst til å gjøre livet vårt litt enklere?

Vi skal nå lage en subklasse `FancySafeArray` av `SafeArray`, som løser de forrige kravene, og noen nye:

1. `FancySafeArray` skal kunne inneholde de aller fleste datatyper som kan default-konstrueres (mildere krav enn for SafeArray)
2. Det skal være mulig å legge sammen to `FancySafeArray` med +, resultatet av å legge sammen to `FancySafeArray` (A + B), skal være et nytt `FancySafeArray` der alle elementene i A ligger først, og så alle elementene i B: (1, 2) + (3, 4) = (1, 2, 3, 4).
3. Det skal ikke være mulig for +-operatoren å endre på noen av de objektene den tar inn.
4. Det skal være mulig å skrive ut et `FancySafeArray` med <<-operatoren.
5. `FancySafeArray` skal kaste exceptions hvis man forsøker å utføre ulovlige handlinger (Dette vil bli spesifisert nøyere lengre ned).
6. `FancySafeArray` skal IKKE være friend med noe som helst

For testingen sin del, kan det være nyttig å gjenta testene som ble gjort for `SafeArray` for `FancySafeArray`.

a) Opprett klassen `FancySafeArray` som arver fra `SafeArray`

b) Skriv den koden som trengs for å oppfylle krav 4
Du står fritt til hvordan du vil formatere utskriften, men det kan være fordelaktig å skille elementer med et eller annet tegn (f.eks. komma eller mellomrom), og kanskje å ha en eller annen form for skilletegn rundt hele utskriften. (F.eks. parentes, slik at det blir lettere å skille utskrift av flere lister fra hverandre)

NB: Hvilken begrensning legger dette på hva slags elementer vi kan legge inn i tabellen vår? Forklar hvorfor krav 1 sier «de aller fleste» og ikke «alle datatyper» som i Oppgave 1.

c) Skriv de(n) medlemsfunksjon(ene) som trengs for å oppfylle krav 2 og 3
Sørg også for at dette testes i main().
NB: For å oppfylle krav 3, må du kanskje endre noen funksjonshoder i SafeArray også. *Hint: Du vil ha bruk for å kunne bruke [] både på const-SafeArrays, og ikke-const-SafeArrays*

3 Unntak/Exceptions (10%)

Vi har nå løst alle kravene i de foregående oppgavene, med unntak av det siste kravet om exceptions.

a) Et sted vi kan få problemer, er dersom vi forsøker å få tilgang til et element som ikke eksisterer, vi har definert at vi skal kunne aksessere elementer med [], som potensielt kan gi oss problemer hvis vi prøver å be om element 150 i et (Fancy)SafeArray med 10 elementer.
Endre implementasjonen av []-operatoren(e) slik at den kaster en `std::out_of_range`-exception hvis man prøver å aksessere et element som ikke er i tabellen.
Utvid også koden i main() til å overpøve at dette faktisk skjer. (Du må med andre ord legge til try-catch).

b) Et annet potensielt problem er at man prøver å endre størrelsen på tabellen til en negativ størrelse, det kan vi ikke få til å gi mening, så det vil vi helst unngå.
Endre implementasjonen av funksjonen(e) som lar deg endre størrelsen på tabellen, slik at disse nå kaster en `std::out_of_range`-exception.
Finnes det en annen måte å løse dette på?

Del 2: Sudoku

4 Representasjon av et sudoku-brett (20%)

Sudoku er et spill der man skal fylle et 9×9 stort brett med tall slik at hver rad, kolonne og alle 9 bokser med størrelse 3×3 inneholder tallene 1-9 nøyaktig én gang.

		n^2								
		n			n			n		
n^2	n							4		8
						1	9			2
					4	3			9	5
	n			6		5				9
			1	5	9				6	
			3				8	5		4
	n	1					6			
				2		4				
		4	6	8	5		3			

Tabell 1: Et vanlig sudoku-brett av størrelse 9×9 med 3×3 ruter, dvs. med $n = 3$ og $n^2 = 9$.

Det endelige målet med øvingen er å lage et program som tar inn et uløst generalisert sudoku-brett og forsøker å løse det. Før du begynner på oppgaven kan det være lurt å lese gjennom artikkelen om sudoku på Wikipedia. For å kunne nå målet vårt om å lage et program som løser et sudoku-brett, vil vi være nødt til å lage følgende:

1. En klasse for å representere brettet og inneholde funksjonene som løser brettet.
2. Funksjoner for å lese inn et brett fra fil.
3. Funksjoner for å skrive resultatet til fil eller skjerm.
4. Funksjoner for å løse brettet.
5. Funksjoner som lar brukeren hjelpe løsningen på vei når den setter seg fast.

I denne første oppgaven skal vi utføre de første tre punktene.

a) Tenk over følgende:

- Hvordan kan et sudokubrett representeres på best mulig måte?
- Hvordan er det smart å representere de tomme rutene på brettet?

b) Opprett en klasse til formålet over, med alle nødvendige medlemsvariabler, konstruktører og destruktør.

c) Vi ønsker å kunne lese, og skrive sudokubrett fra fil:

- **Skriv en funksjon som skriver et (delvis) løst Sudoku-brett til tekstfil.**
  Hint: Er det en måte vi kan gjøre dette på slik at vi også kan bruke samme funksjon til å skrive brettet til skjerm?
- **Skriv en funksjon som leser et Sudoku-brett fra tekstfil.**

Siden mange taktikker for løsing av Sudoku går ut på å finne ut hvilke tall som er mulig å plassere i en gitt rute, kan det være hensiktsmessig å holde orden på dette slik at vi ikke trenger å regne ut dette på nytt igjen hele tiden.

For å gjøre dette kan du for eksempel bruke 9 `std::set` for kolonnene, 9 for radene og 9 for boksene. Da vil tilgjengelige tall i en gitt rute være gitt av snittet (på engelsk *set intersection*) mellom settene for kolonnen, raden, og boksen ruten er i. (Med andre ord de tallene som finnes i alle tre settene).

d) Legg til tre `std::set` som beskrevet over i klassen, og sørg for at de initialiseres riktig i konstruktørene.

e) Avhengig av hvordan du har valgt å oppbevare settene for 3×3-boksene, trenger du å vite hvilket set som tilhører en gitt rute. Du har her to valg: du kan enten ha laget deg en tabell på 9 elementer av type `std::set` (og valgt en eller annen numerering av rutene), eller du kan ha laget deg en `std::set`[3][3]-tabell.

Uansett hvordan du har løst det trenger du funksjonalitet for å finne riktig boks gitt koordinatene til en rute. **Skriv funksjon(er) som kan brukes for å finne en boks, gitt koordinatene til en rute.**

f) Vi er også interessert i at settene skal bli oppdatert riktig når vi leser inn et brett. **Legg til en funksjon som tar inn koordinater og en verdi og setter inn verdien på brettet.** Husk å oppdatere settene tilsvarende.

g) Oppdater funksjonen som leser brettet fra fil, slik at denne bruker funksjonen fra forrige oppgave til å sette verdiene inn på brettet.

5 Unntakshåndtering (15%)

Det er flere ting som kan gå galt i Sudoku-klassen vår, blant annet i forbindelse med lesing og skriving av brett til fil. Det hadde vært nyttig å kunne vite når det skjer noe galt og å kunne håndtere eventuelle feil som skulle skje under kjøring av programmet vårt. I denne oppgaven skal vi bruke unntak (på engelsk *exceptions*) til dette.

a) Lag en klasse `IllegalSudokuPositionException` som arver fra `std::exception`.

Klassen skal brukes til å kaste unntak dersom vi forsøker å plassere et tall i en ugyldig posisjon på brettet. Et tall er i en ugyldig posisjon dersom det samme tallet allerede eksisterer i samme kolonne, rad, eller boks. Klassen skal kunne inneholde posisjonen vi prøvde å plassere tallet i (koordinater på brettet), tallets verdi, og en måte å indikere hvorvidt det er rad, kolonne, eller boks som skaper problemer for tallet. Dersom det er flere årsaker, holder det å indikere én av dem.

b) Lag en klasse `NumberAlreadyInPositionException` som arver fra `std::exception`.

Klassen skal brukes til å kaste unntak dersom vi forsøker å sette inn et tall i en rute som ikke er tom. Klassen skal kunne inneholde posisjonen vi prøvde å plassere tallet i og verdien til tallet som er i den posisjonen fra før.

c) Skriv kode i funksjonen din fra deloppgave 1 f) som sjekker hvorvidt tallet kan plasseres i ruten. Dersom tallet ikke kan plasseres i ruten, skal du kaste et av de to unntakene vi nettopp skrev avhengig av hva som var problemet.

d) Lag en klasse `CouldNotOpenSudokuFileException` som arver fra `std::runtime_error`.

Klassen skal brukes til å kaste unntak dersom vi forsøker å åpne en Sudoku-brettfil og vi ikke klarer det (for eksempel dersom filen ikke finnes). Klassen skal inneholde navnet på filen vi ikke klarte å åpne.

e) Legg til kode i fillesningsfunksjonen din som kaster unntaket vi nettopp lagde dersom filen ikke lar seg åpne.

Vi har nå laget tre egne unntak og lagt til kode for å kaste disse når det er nødvendig. Unntak er imidlertid ikke så nyttige hvis vi ikke fanger dem opp og prøver å løse problemet som førte til unntakene. Vi skal nå bruke `try-catch`-blokker til å fange unntak.

f) Legg til kode i Sudoku-klassen som lar brukeren skrive inn et nytt filnavn å lese et Sudoku-brett fra dersom den første filen ikke lot seg åpne. Gjør dette ved å fange `CouldNotOpenSudokuFileException`.

g) **Legg til kode i Sudoku-klassen som fanger `NumberAlreadyInPositionException` og `CouldNotOpenSudokuFileException` og gir brukeren en fornuftig feilmelding som forklarer hva som er galt.**

6 Spille Sudoku (20%)

a) **Lag en funksjon som finner snittet (på engelsk *set intersection*) av 3 sett av den typen du bruker til å holde orden på tilgjengelige tall.**

b) **Lag en funksjon som tar inn koordinater og en verdi og ved hjelp av settene finner ut hvorvidt verdien er lov å plassere i ruten.**

c) **Lag en funksjon som spør brukeren etter en verdi og koordinatene, og forsøker å plassere denne verdien i de gitte koordinatene. Funksjonen skal la brukeren fortsette å angi nye verdier og koordinater til brukeren velger å ikke angi flere.** Bruk unntakene vi lagde i oppgave 2 til å håndtere tilfellet der brukeren skriver inn ugyldige koordinater, og fang dem opp slik at han kan prøve å skrive inn koordinater på nytt.

d) Vi skal nå implementere funksjonalitet for å la brukeren angre trekkene sine.

- **Legg til en måte å lagre alle trekk som blir gjort i klassen, og skriv om funksjonen fra deloppgave 2 f) slik at alle trekk blir notert ned.**
- **Legg til funksjonalitet i funksjonen fra deloppgave c) som lar brukeren angre et bestemt antall av de siste trekkene.**

e) **Legg til kode i main slik at du kan spille. Test så koden din med et Sudoku-brett, for eksempel det fra starten av oppgave 4.** For å sjekke løsningen din kan du for eksempel bruke denne nettsiden til å løse det samme brettet.

7 Automatisk løsing av sudoku-brett (25%)

a) **Skriv en funksjon som går gjennom hele brettet og for hver rute sjekker hvorvidt det bare er ett tall som kan plasseres i ruten. Hvis det er tilfellet skal tallet plasseres der.**

b) **Skriv en funksjon som forsøker å løse brettet ved å gjenta funksjonen fra forrige oppgave så lenge den klarer å plassere tall. Når funksjonen stopper opp skal programmet spørre brukeren om hvilket trekk den skal gjøre.**.

c) For å løse hele eksempelbrettet vårt trenger vi litt mer funksjonalitet. Vi må nemlig vite hvorvidt et tall kan plasseres et annet sted i samme rad, kollonne eller boks. I motsatt fall er tallet nemlig *nødt* til å plasseres i den ruten vi ser på. Med andre ord: dersom en viss rute er det eneste stedet i enten raden, kolonnen eller 3×3-boksen der tallet kan plasseres, må tallet plasseres der.

- **Skriv de funksjonene du behøver for å finne ut hvor mange steder i en gitt rad et tall kan plasseres.**
- **Skriv de funksjonene du behøver for å finne ut hvor mange steder i en gitt kolonne et tall kan plasseres.**
- **Skriv de funksjonene du behøver for å finne ut hvor mange steder i en gitt boks et tall kan plasseres.**

d) **Sett sammen funksjonene fra forrige oppgave til en funksjon som sjekker om et tall bare kan plasseres i en gitt rute, og bruk denne funksjonen på samme måte som funksjonen fra deloppgave b).** Sjekk også at eksempelbrettet nå kan løses uten brukerhjelp.

NB: Legg merke til forskjellen her, den forrige funksjonen fant ut hvorvidt det bare var ett tall som passet i en gitt rute, mens denne finner ut om ei rute er den eneste som et gitt tall kan plasseres i

e) **(Valgfritt) Skriv kode som genererer et gyldig, uløst Sudoku-brett, og lagrer brettet til fil.** Se for eksempel artiklene Mathematics of Sudoku og Sudoku Puzzles Generating: from Easy to Evil.