



**Frist: 2020-03-27**

### Mål for denne øvingen:

- Bruke beholderne som finnes i Standard Template Library (STL)
- Lære om iterasjoner og bruke dem med beholderne fra STL
- Implementere egne utgaver av noen av disse beholderne
- Lære om templates og bruke disse til å gjøre funksjoner og klasser mer generelle
- Lære om `std::unique_ptr`

### Generelle krav:

- Bruk de eksakte navn og spesifikasjoner gitt i oppgaven.
- Teorioppgaver besvares med kommentarer i kildekoden slik at læringsassistenten enkelt finner svaret ved godkjenning.
- **Denne øvingen skal implementeres uten hjelp fra `std_lib_facilities.h`.**
- 70% av øvingen må godkjennes for at den skal vurderes som bestått.
- Øvingen skal godkjennes av stud.ass. på sal.
- Det anbefales å benytte en programmeringsomgivelse (IDE) slik som Visual Studio Code.

### Anbefalt lesestoff:

- Kapittel 19.3, 19.5 og 20 i PPP.
- Beskrivelsen av beholderne i STL i boka og ekstra detaljer på [cpreference.com](http://cpreference.com)

I denne øvingen skal vi ikke bruke `std_lib_facilities.h` så du må passe på å inkludere eventuelle biblioteker du trenger og holde orden på navnerom selv.

## 1 Iteratorer (15%)

I denne oppgaven skal vi anvende iteratorer på en beholder fra standardbiblioteket som du kjenner fra før, nemlig `std::vector`. Iteratorer er beskrevet i kapittel 20 i boken.

- a) Lag en `std::vector<std::string>` og legg inn en håndfull strenger i vektoren. Skriv ut vektorens innhold med en for-løkke som bruker iteratorer og *ikke* indeksooperatoren (`operator[]`) eller range for-løkker.

Eksempel: Lorem, Ipsum, Dolor, Sit, Amet, Consectetur.

- b) Bruk en reversert iterator (på engelsk *reverse iterator*) for å skrive ut innholdet av vektoren i motsatt rekkefølge.

Eksempel: Consectetur, Amet, Sit, Dolor, Ipsum, Lorem.

- c) Skriv en funksjon `replace` som skal ta inn en referanse til en `std::vector<std::string>` og to `std::string`-variabler `old` og `replacement` som argumenter. Funksjonen skal gå gjennom vektoren og erstatte hvert element den finner som er likt `old` med `replacement`.

For å gjøre dette skal du bruke iteratorer og medlemsfunksjonene `erase()` og `insert()`. Dersom det ikke finnes noen elementer lik `old` i vektoren, skal funksjonen ikke endre på vektoren. Du kan bruke funksjoner fra `<algorithm>` hvis du ønsker det.

Eksempel: Vektoren inneholder i utgangspunktet Lorem, Ipsum, Dolor, Lorem. Vi kjører `replace(vektor, "Lorem", "Latin");`. Vektoren inneholder deretter Latin, Ipsum, Dolor, Latin.

- d) Gjenta oppgave a–c, men gjør det nå for et `std::set`.

Ser du likhetene i måten vi gjør det for et `set`, og for en `vector`?

## 2 Lenkede lister (20%)

Nyttig å vite: `std::forward_list` og `std::list`

I denne oppgaven skal vi studere lenkede lister (*linked list* på engelsk). En lenket liste er en liste hvor hvert element (node), i tillegg til å inneholde data, inneholder en peker til det neste elementet i listen. På samme måte som `set`- og `vector`-beholdere, inneholder C++ sitt standardbibliotek en implementasjon av lenkede lister, nemlig `std::forward_list`, og `std::list`.

`std::forward_list` er en enkelt-lenket liste. Dette betyr at hver node kun har en peker til den neste noden i lista, men *ikke* til den forrige noden. Vi skal *ikke* benytte oss av denne i øvinga. `std::list` er en dobbelt-lenket liste, som vil si at den har en peker til den forrige noden i tillegg til en peker til den neste noden. Dobbelt-lenkede lister er enklere å jobbe med fordi man kan «navigere» både fram og tilbake inni listen, og brukes derfor ofte i praksis.

- a) Lag en klasse **Person** med medlemsvariabler for fornavn og etternavn. Inkluder alle konstruktører, medlemsfunksjoner og overlagrede operatorer du mener er nyttige, inkludert en måte å skrive ut **Person**-objekter til skjermen.
- b) Skriv en funksjon for å sette inn **Person**-objekter i en `std::list` i sortert rekkefølge. Objektene skal være sortert basert på den alfabetiske rekkefølgen til personenes navn. Funksjonen kan for eksempel ha prototypen

```
void insertOrdered(std::list<Person> &l, const Person& p);
```

Test denne funksjonen ved å opprette en variabel av typen `std::list<Person>` og sette inn en rekke **Person**-objekter i listen ved hjelp av `insertOrdered`. Lag så en løkke i `main()` som skriver ut alle objektene i listen til skjermen.

*Hint: Strenger kan sammenlignes alfabetisk ved hjelp av operatorene `<` og `>`, slik at uttrykket `"ABCD" < "BCDEF"` for eksempel er sant.*

### 3 LinkedList (25%)

Vi skal nå implementere vår egen dobbeltlenkede liste. Bruk den vedlagte filen `LinkedList.h` som basis og implementer en lenket liste hvor hver node inneholder en `std::string` som data.

#### Nyttig å vite: `std::unique_ptr`

Nodene (Node-klassen) i `LinkedList` bruker `std::unique_ptr` for å peke på den neste noden. I tillegg har `LinkedList` en `std::unique_ptr` til det første elementet i listen.

**Hva er en `std::unique_ptr`?** `std::unique_ptr` er en smartpeker som eier og håndterer et annet objekt gjennom en peker, og sletter automatisk minnet som tilhører objektet når `std::unique_ptr`-instansen destrueres. Dette vil si at vi slipper å bruke `new` og `delete` når vi skal bruke dynamisk allokerede objekter. Dette faget har kun en liten intro til det man kan gjøre med `std::unique_ptr`.

**Hvordan opprettes en `std::unique_ptr`?** C++14-funksjonen `std::make_unique` brukes for å lage et nytt `std::unique_ptr`-objekt. Den allokerer også nødvendig minne til objektet `std::unique_ptr` håndterer.

```
/* Her opprettes en unique_ptr som håndterer et Student-objekt.
 * Argumentene til funksjonen gis direkte til Student-konstruktøren. */
unique_ptr<Student> s1 = make_unique<Student>("daso", "daso@stud.ntnu.no");
/* Vi kan også bruke auto når vi oppretter en unique_ptr. */
auto s2 = make_unique<Student>("lana", "lana@stud.ntnu.no");
```

**Hvordan brukes en `std::unique_ptr`?** Derefereringsoperatoren (\*) og piloperatoren (->) brukes som om det er en vanlig peker.

```
cout << s1->getName() << '\n';
cout << *s2 << '\n';
```

En `unique_ptr` eier objektet det peker til (bare en `unique_ptr` kan peke på et objekt om gangen) — den tillater ikke pekeren å bli kopiert, men eierskapet kan overføres vha. `std::move`. Vi sier at `std::move` overfører eierskapet av objektet.

```
/* Overfør eierskap fra s1 til s3. */
auto s3 = move(s1); /* Verdien til s1 er nå udefinert. */
/* Kodelinjen under vil ikke kompilere, for unique_ptr kan ikke kopieres*/
// auto s3 = s1;
```

Noen ganger ønsker vi å la andre bruke objektet som blir pekt til av `unique_ptr`-instansen uten å overføre eierskapet. Vi kan da få tak i den underliggende pekeren, vha. medlemsfunksjonen `get`.

```
void printStudent(Student* sPtr);
printStudent(s2.get()); /* s2.get() returnerer den underliggende pekeren.*/
```

Medlemsfunksjonen `get` kan også brukes til å sjekke om en `unique_ptr` har et tilknyttet objekt, ved å sammenlikne med `nullptr`.

```
if (s1.get() != nullptr) {
    cout << "S1 contains an object\n";
} else {
    cout << "S1 does not contain an object\n"; // <- Dette skrives ut.
}
if (s3.get() != nullptr) {
    cout << "S3 contains an object\n"; // <- Dette skrives ut.
} else {
    cout << "S3 does not contain an object\n";
}
```

Den utdelte koden har deklartert to klasser: `Node` og `LinkedList`. Hensikten med `Node` er at hvert objekt av klassen skal representere ett element i den lenkede listen `LinkedList`. Hvert `Node`-objekt har en `string`-verdi `value`, en `unique_ptr` `next` til det neste `Node`-objektet i `LinkedList` og en vanlig peker `prev` til det forrige `Node`-objektet i `LinkedList`. Videre har `Node` konstruktører (én default og en der medlemsvariablene settes lik input-argumentene), én default-destruktør og get-funksjoner for medlemsvariablene. `Node` har også satt operatoroverlastingen til « og klassen `LinkedList` som `friend`.

`LinkedList` har de private medlemmene `head`, som er en `unique_ptr` til det første elementet i lista, og `tail`, som er en vanlig peker som peker "forbi" siste element/peker til en tom node som viser at vi er på enden av lista. Konstruktøren, destruktøren og funksjonene `isEmpty`, `begin` og `end` er allerede definert. Oppgaven din er å definere de resterende funksjonene, slik at `LinkedList`-klassen får samme oppførsel som datastrukturen til en lenket liste. Hva de forskjellige funksjonene skal gjøre står beskrevet i `LinkedList.h`.

**a) Implementer følgende funksjoner og operator:**

**Tenk over følgende:** Hvorfor er medlemsvariabelen `prev` i `Node` av typen `Node*`, og ikke en `std::unique_ptr`?

- `std::ostream& operator<<(std::ostream& os, const Node& node)`
- `Node* LinkedList::insert(Node* pos, const std::string& value)`
- `Node* LinkedList::remove(Node* pos)`
- `Node* LinkedList::find(const std::string& value)`
- `void LinkedList::remove(const std::string& value)`
- `std::ostream& operator<<(std::ostream& os, const LinkedList& list)`

Les beskrivelsene av hvordan funksjonene skal fungere i `LinkedList.h`.

**b) Svar på følgende teorispørsmål:**

- Du har i de tidligere øvingene hovedsakelig brukt `std::vector` som beholder. Imidlertid vil det i noen tilfeller være et bedre valg å bruke en lenket liste. Når er lenkede lister bedre, og hvorfor? *Hint: Tenk på hvor lang tid det tar å utføre vanlige operasjoner i en lenket liste og i en `std::vector`.*
- Den lenkede listen du nettopp lagde kan brukes til å implementere andre datastrukturer på en lett måte. Forklar hvordan ville du brukt `LinkedList` klassen for å implementere en *stack* eller *queue*. (Du skal *ikke* å implementere dem.) Disse datastrukturene er også beskrevet i boken.

## 4 Templates for funksjoner (20%)

Templates er en form for *generisk programmering* som lar oss skrive generelle funksjoner som fungerer for mer enn én datatype uten å tvinge oss til å lage separate implementasjoner for hver enkelt datatype. I denne oppgaven skal vi skrive noen slike. Templates er beskrevet i kapittel 19.3 i boken.

**a) Skriv template-funksjonen `maximum` som tar inn to verdier av samme type som argument og returnerer den største verdien av de to.** Funksjonen skal være skrevet slik at følgende kode skal compilere uten feil og gi forventede resultater ved kjøring.

```
int a = 1;
int b = 2;
int c = maximum(a, b);
// c er nå 2.
```

```
double d = 2.4;
double e = 3.2;
double f = maximum(d,e);
// f er nå 3.2
```

Denne funksjonen vil fungere for alle grunnleggende datatyper (for eksempel `int`, `char` og `double`), men hvis du bruker argumenter av en egendefinert type, som en `Person`- eller `Circle`-klasse, er sjansen stor for at koden din ikke vil kompilere. Hvorfor? Hva må du gjøre for å kunne bruke denne funksjonen med objekter av andre typer?

- b) **Skriv template-funksjonen `shuffle` som stokker om på elementene i en `vector` slik at rekkefølgen på elementene i `vector`-en blir tilfeldig..**

Funksjonen skal være skrevet slik at følgende kode skal kompilere uten feil og gi forventede resultater ved kjøring.

```
vector<int> a{1, 2, 3, 4, 5, 6, 7};
shuffle(a); // Resultat, rekkefølgen i a er endret.

vector<double> b{1.2, 2.2, 3.2, 4.2};
shuffle(b);

vector<string> c{"one", "two", "three", "four"};
shuffle(c); // Resultat, rekkefølgen i c er endret.
```

## 5 Templates for klasser (20%)

I tillegg til å kunne brukes til å gjøre funksjoner mer generelle, kan vi også benytte templates til å generalisere klasser. Eksempler på dette finner man blant annet i STL: `std::vector`, `std::set` og alle de andre beholderne i STL er implementert ved hjelp av templates.

I denne oppgaven skal koden fra `LinkedList` utvides til å benytte templates. Før du begynner på oppgaven er det viktig at all koden for klassen, inkludert implementasjonen av medlemsfunksjonene, befinner seg i en headerfil. Dette er fordi kompilatoren må kjenne til hele klassen (både deklarasjon og implementasjon) for å kunne generere riktig spesialisering av klassen hver gang den benyttes.

- Omskriv `LinkedList` slik at den ved hjelp av klasse-templates kan håndtere vilkårlige typer, ikke bare strenger.
- Er det noen spesielle hensyn som må tas for at en datatype skal kunne brukes med `LinkedList`?