Norges teknisk–naturvitenskapelige universitet
Institutt for datateknologi og informatikk

TDT4102 Prosedyre- og objektorientert programmering
Vår 2022

Øving 11

Frist: 2021-04-08

Mål for denne øvingen:

- Bruke beholderne som finnes i standardbiblioteket
- Lære om iteratorer og bruke dem med beholderne fra standardbiblioteket
- 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 standardbiblioteket i boka og ekstra detaljer på cppreference.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* indeksoperatoren (`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`?

Nyttig å vite: Navnerom

En navnekonflikt oppstår hvis flere forskjellige ting har samme navn. Siden et navn refererer til flere ting vet ikke kompilatoren hva den skal gjøre. Navnerom (`namespace`) brukes for forhindre slike navnekonflikter.
Et eksempel på hva som ville skjedd uten navnerom: Standardbiblioteket inneholder beholderen `vector`. Dersom vi ønsket å lage vår egen matematiske vektor kunne vi ikke kalt den `vector` uten navnerom.

```
#include <vector> // Inkluderer beholderen vector fra standardbiblioteket
 // Definerer vår egen matematiske vector
class vector { double x, y; };
int main() {
    vector v; // Hvilken vector er dette?
}
```

Et eksempel på slik det faktisk er: For å unngå slike navnekonflikter defineres alle navn i standardbiblioteket i navnerommet `std`.

```
#include <vector> // Inkluderer beholderen vector fra standardbiblioteket
 // Definerer vår egen matematiske vector
```

```
class vector { double x, y; };
int main() {
    std::vector v; // Beholderen fra standardbiblioteket
    vector u; // Vår matematiske vector
}
```

Vi kan også definere vårt eget navnerom for å skille navnene fra hverandre:

```
#include <vector> // Inkluderer beholderen vector fra standardbiblioteket
namespace math { // Definerer vårt eget navnerom
 // Definerer vår egen matematiske vector
class vector { double x, y; };
} // namespace math slutt
int main() {
    std::vector v; // Beholderen fra standardbiblioteket
    math::vector u; // Vår matematiske vector
}
```

Dersom det ikke er noen fare for en navnekonflikt kan vi hente navnene fra et navnerom inn i det globale navnerommet ved å bruke `using namespace`.

```
#include <iostream> // std::cout
int main() {
    // Må skrive std:: for å fortelle hvilket navnerom cout er i
    std::cout << "Hello world!\n";
}
```

```
#include <iostream> // std::cout
using namespace std; // Henter alle navnene i std inn i det globale navnerommet
int main() {
    cout << "Hello world!\n"; // Kan droppe std::
}
```

Det er generelt frarådet å bruke `using namespace` i headerfiler (CPP Core Guideline SF.7).

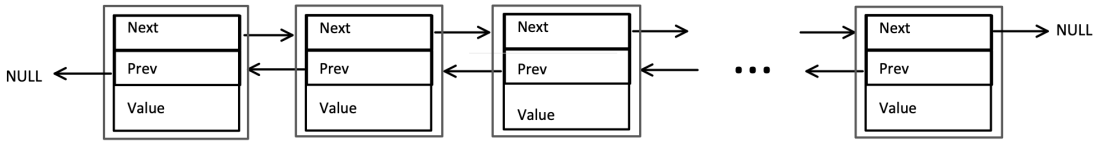
2 Lenkede lister (20%)

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. En grafisk representasjon av en dobbelt-lenket liste er vist under. Som man ser fra figuren har hver node altså to pekere i tillegg til den faktiske verdien til noden. `std::list` er laget ved hjelp av templates, så nodenes verdi kan ha vilkårlig type. Første og siste node har nullpekere der de andre nodene har pekere til henholdsvis forrige og neste node. Den innebygde dobbelt-lenkede listen `std::list` har funksjonene `begin()` og `end()` som returnerer iteratorer til henholdsvis første node og til en tom node etter siste node. 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: Hva skjer hvis du legger til en person i en tom liste og på slutten av en liste?

3 LinkedList (25%)

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

Den utdelte koden har deklarert 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 `operator<<` 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`.
Hint: Det kan lønne seg å tegne opp en lenket liste med pekerne mellom nodene og se hvordan pekerne må flyttes for å legge til og fjerne elementer.
Hint: Pass på edge-cases som blant annet å legge til elementer før det første elementet eller å fjerne det første elementet i listen.

Nyttig å vite: plassering av `const`

I den utdelte koden har LinkedList en medlemsvariabel som ser slik ut

```
Node* const tail;
```

Dette kan se litt mystisk når man er vant til å se `const` før typen til variabelen, f.eks.

```
const Node* a;
```

Forskjellen er at i det første eksempelet lager vi en konstant peker til en variabel av typen Node (som ikke er const), mens i det andre har vi en peker (som ikke er konstant) til en variabel som er const. I første eksempel har altså en peker som ikke kan endre seg, men alltid vil peke på samme sted i minnet. Vi kan likevel endre det som ligger lagret der. Eksempel 2 har en peker der vi kan endre hvor den peker, men variabelen den peker på kan ikke endres.

Regelen er at `const` påvirker det til venstre for seg, med mindre det ikke er noe der, da påvirker den det til høyre. Hvis det blir forvirrende kan man prøve å lese linjer med const i fra høyre til venstre, da vil for eksempel dette

```
int const * returnConstNumPointer() const;
```

være en **konstant** funksjon `returnConstNumPointer()` som returnerer en peker (*) til et **konstant** heltall (int).

Hvis du er interessert kan du lese mer om const her
(Spesielt helt nederst under overskriften *East Const, Const West.*)

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.

Nyttig å vite: Template

Dere er kjent med å overlagre funksjoner fra tidligere. For eksempel kan man definere funksjonen `add()` for både `int` og `double`.

```
int add(int lhs, int rhs){
    return lhs + rhs;
}
double add(double lhs, double rhs){
    return lhs + rhs;
}
int main() {
    add(1, 2); // Gir 3
    add(1.0, 2.0); // Gir 3.0
}
```

Du legger kanskje merke til at definisjonen av de to funkjsonene er helt like. Eneste forskjellen er typene til parameterne og returverdien. Med templates kan disse kombineres til en og man slipper duplisering av koden.

```
template<typename T>
T add(T lhs, T rhs){
    return lhs + rhs;
}
int main() {
    add(1, 2); // Gir 3
    add<int>(1, 2); // Eksplisitt template-argument, samme som over
    add(1.0, 2.0); // Gir 3.0
    add<double>(1.0, 2.0); // Eksplisitt template-argument, samme som over
}
```

Dere er allerede kjent med å bruke templates og syntaksen for å spesifisere template-argument fra bruk av `std::vector`.

```
std::vector<int> i; // int er template-argument, gir en vektor med heltall
std::vector<double> d; // int er template-argument, gir en vektor med flyttall
```

Det er også mulig å definere klasse-templates:

```
template<typename T, typename U>
struct Pair{
    T first;
    U second;
};
```

Her har vi definert en type med to medlemsvariabler som kan være av hvilken som helst type.

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 kompilere 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); // Resultat, rekkefølgen i b er endret.

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 standardbiblioteket: `std::vector`, `std::set` og alle de andre beholderne i standardbiblioteket er implementert ved hjelp av templates.

I denne oppgaven skal koden fra `LinkedList` utvides til å benytte templates.
Merk: 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.

a) Omskriv `LinkedList` slik at den ved hjelp av klasse-templates kan håndtere vilkårlige typer, ikke bare strenger.

b) Er det noen spesielle hensyn som må tas for at en datatype skal kunne brukes med `LinkedList`?++ Hint: Tenk over hvilke operasjoner som gjøres på elementene i listen.

6 Frivillig: Et problem med destruktøren til `LinkedList`

I denne øvingen har vi brukt `std::unique_ptr` slik at vi ikke trenger å holde orden på minne selv. Derfor har vi ikke trengt å definere destruktøren.

Nå skal vi se på et problem som skjer hvis vi har mange elementer i listen. Kopier funksjonen under inn i prosjektet ditt og kall den fra `main()`. Her lager vi en liste med mange elementer før listen går ut av skop og destruktøren kjører.

```
void testLinkedListOverflow() {
    std::cout << "Testing LinkedList destructor overflow\n";

    using LinkedList::LinkedList;
    constexpr unsigned int N = 1'000'000;
    {
        LinkedList list;
        for (unsigned int i = 0; i < N; ++i) {
            list.insert(list.end(), "");
        }
        std::cout << "Elements added\n";
    } // Destruktøren kjører
    std::cout << "List destructed\n";
}
```

Med tilstrekkelig mange elementer i listen vil man få en «stack overflow» når destruktøren kjører. Kjør koden med debuggeren og se i call-stacken hva som skjer. Ser du problemet?

Når destruktøren til `LinkedList` kjører må destruktøren til `head` kjøres. `head` er av typen `std::unique_ptr<Node>`. Hver `Node` har en medlemsvariabel `next` av typen `std::unique_ptr<Node>`, så når destruktøren til `Node` kjøres må destruktøren til en annen node kjøres i tillegg. Dette fortsetter for hver node i listen. For at første element i listen skal destrueres må andre element i listen destrueres. For at andre element i listen skal destrueres må tredje element i listen destrueres, osv. Det er først når siste element i listen er destruert at alle destruktørkallene kan avsluttes. Med mange nok elementer i listen vil man gå tom for plass i stacken og man får en såkalt «stack overflow».

a) Implementer en iterativ destruktør for `LinkedList` Den nåværende rekursive destruktøren fungerer ikke for lange lister. Implementer en destruktør som «kobler av» ett

og ett element fra slutten av listen. Dette gjør at kun en `Node`-destruktør kjører av gangen og vi unngår stack overflow. Kjør testfunksjonen over igjen og se at du ikke lenger får noen stack overflow selv for store lister.