



Norges teknisk-naturvitenskapelige
universitet
Institutt for datateknikk og
informasjonsvitenskap

TDT4102 Prosedyre-
og objektorientert
programmering
Vår 2016

Øving 11

Frist: 2016-04-15

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

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.

Anbefalt lesestoff:

- Kapittel 16, 17 og 19, Absolute C++ (Walter Savitch)
- Beskrivelsen av beholderne i standardbiblioteket til C++ på cppreference.com

1 Iteratorer (10%)

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 19.1 i boken.

- Lag en `std::vector<string>` og legg inn en håndfull strenger i vektoren. Skriv ut vektorens innhold med en for-løkke som bruker iteratorer og *ikke* indeksope-
ratoren `[]`. Eksempel: Lorem, Ipsum, Dolor, Sit, Amet, Consectetur.
- Bruk en reversert iterator (på engelsk *reverse iterator*) for å skrive ut innhol-
det av vektoren i motsatt rekkefølge. Eksempel: Consectetur, Amet, Sit, Dolor,
Ipsum, Lorem.
- Skriv en funksjon `replace` som skal ta inn en referanse til en `std::vector<string>`
og to `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.

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

2 set-beholdere (20%)

Vi har så langt i øvingsopplegget sett på to beholdere (på engelsk *containers*) fra standard-
biblioteket til C++, `std::vector` og `std::map`. Disse beholderne er en del av den biten av
standardbiblioteket som kalles for «Standard Template Library» (STL), som også inneholder en
rekke andre beholdere. I denne oppgaven skal vi først se på bruk av beholderen `std::set`, før
vi implementerer vår egen utgave av denne beholderen.

`std::set` er en implementasjon av det som i matematikken kalles en mengde eller *set* på engelsk,
altså en samling av elementer der hvert element kun finnes én gang og der elementene ikke er i
noen spesiell rekkefølge. Overført til dataverden blir et `set` en datastruktur som tillater følgende
operasjoner:

- `insert`: Sett et element inn i settet
- `remove`: Fjern et element fra settet
- `exists`: Sjekk om et element finnes i settet

I tillegg er altså alle elementer i et `set` unike. Det vil si at om man legger inn et element som
allerede finnes i settet skal det fortsatt kun finnes ett eksemplar av elementet i settet. I denne
oppgaven skal vi se på `set` som inneholder heltall (`int`).

- Lag et `std::set` som inneholder alle heltallene fra 0 til og med 100. Husk at
du må inkludere headerfilen `<set>` for å kunne bruke `std::set`. Etter at du har lagt til
tallene, prøv å legge dem til samme beholder på nytt. Hva skjer?
- Fjern alle tall som er delbare med 2 fra settet, med unntak av tallet 2 selv.
Skriv ut alle verdiene i settet ved hjelp av en iterator.
- Gjør det samme som i forrige deloppgave for hvert tall mellom 2 og 50. Fjern
altså alle tall som er delelige på 3 utenom 3 selv, alle tall som er delelige på 4 utenom 4
selv, og så videre. Skriv igjen ut all verdiene i settet. Hva kalles tallene vi sitter igjen med?

Vi skal nå lage vår egen `set`-beholder som implementerer oppførselen beskrevet på starten av oppgaven. Denne skal vi kalle `SimpleSet`. I motsetning til i tidligere øvinger, da du skrev spesifikasjoner for klassene dine i headerfilen og selve implementasjonen i `.cpp`-filen, bør du denne gangen skrive hele klassen i headerfilen. Noe annet vil føre til problemer senere når vi skal utvide klassen til å bruke templates.

- d) **Implementer klassen `SimpleSet`.** For å gjøre dette enklere for deg selv, kan du bruke spesifikasjonen gitt i den vedlagte filen `SimpleSet.h`. I tillegg til oppsettet fra denne filen trenger du en måte å skrive ut elementene i settet på. I stedet for å lage din egen iterator (kan være komplisert) kan du for eksempel lage en `operator <<` som skriver ut elementene.
- e) **Gjenta de tre første deloppgavene, men bruk nå `SimpleSet` i stedet for `std::set`.** Sjekk at klassen din oppfører seg på samme måte som `std::set`.

3 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 16 i boken.

- a) **Skriv template-funksjonen `shuffle` som stokker om på elementene i en tabell (array) slik at rekkefølgen på elementene i tabellen blir tilfeldig.** La tabellen som skal stokkes være den første parameteren og størrelsen på tabellen den andre.

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

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

```
double b[] = {1.2, 2.2, 3.2, 4.2};
shuffle(b, 4);
```

```
string c[] = {"one", "two", "three", "four"};
shuffle(c, 4); // Resultat, rekkefølgen i c er endret.
```

- b) **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 naa 2.
```

```
double d = 2.4;
double e = 3.2;
double f = maximum(d,e);
// f er naa 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 compilere. Hvorfor? Hva må du gjøre for å kunne bruke denne funksjonen med objekter av andre typer?

4 Templates for klasser (10%)

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

5 Lenkede lister (40%)

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, peker til det neste elementet i listen. Som med `set`-beholdere inneholder C++ sitt standardbibliotek en implementasjon av lenkede lister, nemlig `std::forward_list`.

- 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.
- Skriv en funksjon for å sette inn `Person`-objekter i sortert rekkefølge. Objektene skal være sortert basert på den alfabetiske rekkefølgen til personenes navn. Funksjonen kan for eksempel ha prototypen

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

Test denne funksjonen ved å opprette en variabel av typen `std::forward_list<Person>` og sette inn en rekke `Person`-objekter i listen ved hjelp av `insertOrdered`.

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

- Lag en løkke i `main()` som skriver ut alle objektene i listen til skjermen.

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

- Implementer følgende funksjoner og operator:

- `ListNode::ListNode(const string &value)`
- initialiserer noden.
- `string ListNode::getValue() const`
- returnerer verdien til noden.
- `ListNode *ListNode::getNext() const`
- returnerer pekeren til den neste noden.
- `LinkedList::LinkedList()`
- konstruktør som lager en ny liste med head og last satt til `NULL`.
- `LinkedList::~~LinkedList()`
- destruktør som sletter alle elementene i listen for å frigjøre minne.

- `bool LinkedList::isEmpty() const`
- returnerer true hvis listen er tom.
- `void LinkedList::insertAtFront(const string &value)`
- som legger til et element fremst i listen.
- `void LinkedList::insertAtBack(const string &value)`
- som legger til et element bakerst i listen.
- `bool LinkedList::removeFromFront(string &value)`
- som fjerner et element fremst i listen.
Verdien til elementet som fjernes skal lagres i string-referansen som gies som argument.
- `bool LinkedList::removeFromBack(string &value)`
- som fjerner et element bakerst i listen.
Verdien til elementet som fjernes skal lagres i string-referansen som gies som argument.
- `ostream& operator<<(ostream &stream, const &LinkedList)`
- som skriver ut listen i et lesbart format

e) Legg til en medlemsfunksjon for å søke i den lenkede listen etter en gitt **string-verdi**. Funksjonen skal returnere en peker til den første noden som inneholder verdien, eller NULL hvis verdien ikke finnes i listen.

```
ListNode *LinkedList::search(const string &value);
```

f) Legg til en medlemsfunksjon som fjerner og sletter (**delete**) alle elementene i en lenket liste som har lik verdi som en gitt string (argumentet til funksjonen). Pass på at du bruker `removeFromFront` og `removeFromBack` der det er mulig.

```
void LinkedList::remove(const string &value);
```

g) Svar på følgende teorispørsmål:

- Du har i de tidligere øvingene hovedsakelig brukt tabeller (arrays) og `std::vector` som beholdere. 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 tabell eller `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 trenger ikke å implementere dem.) Disse datastrukturene er også beskrevet i boken.

I `std::forward_list` og i vår egen `LinkedList` peker hver node til den *neste* noden i listen. En annen type lenket liste er *dobbelt-lenkede lister*, der hver node både peker til *neste* og *forrige* node i listen. Dobbel-lenkede lister er enklere å jobbe med fordi man kan «navigere» både fram og tilbake inni listen, og brukes derfor ofte i praksis.

h) STL inneholder en implementasjon av dobbelt-lenkede lister som heter `std::list`. **Implementer funksjonen fra deloppgave b) for bruk med `std::list`.**

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

Hvordan kan denne forenkles sammenliknet med funksjonen du skrev for `std::forward_list`? 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`.

i) (Valgfritt) **Skriv om `LinkedList` til en dobbelt-lenket liste.** Bruk også de nye mulighetene i en dobbelt-lenket liste til å forenkle medlemsfunksjonen `remove`.