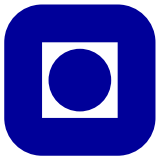

Norges teknisk–naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

TDT4102 Prosedyre- og objektorientert programmering Vår 2017

Øving 11

Frist: 2017-04-07

Mål for denne øvingen:

- Bruke beholderne som finnes i Standard Template Library (STL)
- Lære om iteratorer 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.

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[]`).

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.

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

2 `set`-beholdere (20%)

Nyttig å vite: `std::set`

Vi har så langt i øvingsopplegget sett på to beholdere (på engelsk *containers*) fra standardbiblioteket 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 `std::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 `std::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 `std::set` som inneholder heltall (`int`).

a) 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?

b) 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.

c) 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 kompilere 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 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?

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.

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

b) Er det noen spesielle hensyn som må tas for at en datatype skal kunne brukes med `SimpleSet`?

5 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, peker til det neste elementet i listen. Som med `set`-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 hvert node kun har en peker til den neste noden i lista. `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::forward_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::forward_list<Person> &l, const 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`. 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.

c) Implementer funksjonen fra deloppgave b) for bruk med `std::list`.

```
void insertOrdered(std::list<Person> &l, const 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`.

6 LinkedList (20%)

Vi skal nå implementere vår egen foroverlenkede 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 tilegg 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 allokerte 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.

```
/* Here a unique_ptr is created. This unique_ptr handles a Student-object.
 * The arguments given to make_unique is passed to the Student-constructor */
unique_ptr<Student> s1 = make_unique<Student>("daso", "daso@stud.ntnu.no");
/* auto may also be used when creating new unique_ptrs */
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 — den tillater ikke pekeren å bli kopiert, men eierskapet kan overføres vha. `std::move`. Vi sier at `std::move` *overfører eierskapet* av objeketet.

```
/* create unique_ptr using make_unique - the preferred way */
/* transfer ownership of unique ptr */
auto s3 = move(s1); /* value of s1 is now unspecified */

/* This code won't compile, because you can't copy unique_ptr */
// auto s3 = s1;
```

Noen ganger ønsker vi å la andre bruke objeketet 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() returns the underlying raw pointer */
```

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

```
// s1 was moved from, to s3
if (s1.get() != nullptr) {
    cout << "S1 contains an object\n";
} else {
    cout << "S1 does not contain an object\n"; // <- this will be printed
}

// s3 was moved to, from s1
if (s3.get() != nullptr) {
    cout << "S3 contains an object\n"; // <- this will be printed
} else {
    cout << "S3 does not contain an object\n";
}
```

a) Implementer følgende funksjoner og operator:

- `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& ll)`

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

b) 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.

c) Valgfri: Gjør om `LinkedList` til en dobbeltlenket liste

Du må nå legge til en ny privat medlemsvariabel `prev` i `Node`-klassen. Det kan også være greit med en (`public`) `getPrev`-funksjon.

```
Node* prev;
Node* getPrev() const;
```

Hvorfor er `prev` av typen `Node*`, og ikke en `std::unique_ptr`?

Du må nå endre implementasjonen av konstruktører og medlemsfunksjoner i både `Node` og `LinkedList`. Tenk spesielt over hvordan du kan gjøre `remove` og `insert` raskere nå som du har en bakoverpeker.