Linköpings Universitet
Institutionen för datavetenskap (IDA)
UPP-gruppen 2021-03-22

Standard Template Library

Scenario

Du har blivit anlitad av ett bokförlag för att skriva ett program som hjälper dem att enkelt kunna modifiera texter t.ex. genom att ändra eller ta bort ord. De anställda på bokförlaget är duktiga på att använda Linux-terminalen och har därför bett att programmet ska vara just terminal-baserat. Programmet ska styras genom att skicka kommandoradsargument till programmet.

Vi har två målgrupper för detta program:

- De programmerare som potentiellt ska vidareutveckla programmet i framtiden.
- Förläggare och redaktörer på bokförlaget som använder programmet för att modifiera texter.

När vi använder begreppet *användare* i den här laborationen, då pratar vi om det anställda på förlaget.

Mål

Målet med denna laboration är att använda STL (standardbiblioteket). Framförallt ska du använda databehållare (Engelska: containers), algoritmer, iteratorer och lambdafunktioner.

Det är viktigt att du använder så mycket av standardbiblioteket som du bara kan. Det är bättre att göra för mycket än för lite. Det går att lösa hela den här laborationen utan att ha en enda `for`-, `while`- eller `do-while`-loop i koden. Det går också att lösa utan rekursion. D.v.s. det går att lösa genom att endast använda de komponenter som finns i standardbiblioteket, tillsammans med if-satser, funktioner och undantagshantering.

Du kommer även behöva använda `std::string` och dess inbyggda funktioner, undantag (Engelska: exceptions), kommandoradsargument och strömmar. Undantag används för att rapportera felaktigheter som leder till att programmet inte kan exekveras ordentligt.

Du ska visa på att du kan använda dessa saker på ett *korrekt*, *läsbart* och *skalbart* sätt.

Korrekthet

Det är viktigt att du skriver ditt program på ett sådant sätt att det fungerar oavsett vad användaren (bokförlaget i det här fallet) matar in för något. Om något inte går att utföra ska detta rapporteras till användaren med ett felmeddelande. Detta felmeddelande ska vara förståeligt av personer som inte kan något om programmering, men har koll på terminalen i Linux.

Detta innebär att du måste vara medveten om hur alla standardbibliotekskomponenter du använder fungerar och vad som kan gå fel med dem.

Vid varje steg i din lösning måste du fundera på om det finns någon inmatning användaren kan göra som leder till fel. Fundera också på huruvida det är ett fel som du kan lösa eller om det är ett fel som kan rapporteras till användaren.

Man pratar ofta om användarfel, d.v.s. fel som beror på att användaren gör något felaktigt med programmet. I ditt program ska det vara tydligt när användaren gör ett fel. Programmet får aldrig krascha under körning utan det är ditt ansvar som programmerare att hantera alla tänkbara fel som användaren kan göra.

Det är också ditt ansvar att se till så att alla tänkbara inmatningar fungerar på det förväntade sättet.

Läsbarhet

Koden du producerar måste gå att förstå för en annan C++ programmerare (du kan anta att denne hypotetiska programmerare har läst just den här kursen).

Detta betyder att standardbiblioteket ska användas på det "tänkta" sättet. T.ex. ska algoritmer användas för att lösa det problem som de är designade för att lösa, databehållare ska användas på det sättet de är avsedda för, undantag används för att hantera exceptionella fel (ej programflöden som leder fram till efterfrågad utdata).

Lambdafunktioner får inte vara för långa (runt 1-5 rader räcker). Det ska också vara tydligt exakt vad lambdafunktionen behöver tillgång till för att kunna utföra sitt jobb.

Du ska ha en bra uppdelning av programmet i flera funktioner. Dessa funktioner ska ha bra namn och tydliga syften. Funktioner som gör mer än en sak är oftast mycket svårare att förstå, därför bör du se till att varje funktion endast uppfyller ett syfte.

Kommentarer, där de förekommer, ska beskriva kodens högre syfte: en kommentarer ska beskriva *varför* koden ser ut som den gör, snarare än *hur* koden ser ut (eller *vad* koden gör). En van programmerare kan förstå *vad* koden gör och *hur* den gör det genom att läsa koden, men att förstå *varför* koden ser ut som den gör är mycket svårare.

Kommentarer kan vara bra för att öka läsbarheten, *men* bra skriven kod ska vara självförklarande. Så undvik kommentarer om möjligt, det är bättre att fundera på hur ni kan skriva koden på ett tydligare sätt. Detta är mycket lättare att göra när man jobbar med standardbiblioteket eftersom att komponenterna har väl valda namn.

Skalbarhet

Programmet ska vara enkelt att utöka. Det ska vara enkelt för bokförlaget att anställa en annan programmerare för att lägga till ny funktionalitet i programmet. Detta innebär att koden måste vara skriven på ett sådant sätt att minimal ändring krävs för att lägga till något nytt. I bästa fall ska det

räcka med att *lägga till* ny kod utan att programmeraren behöver ändra något i den befintliga koden.

Att dela upp koden i funktioner och att lägga releterade funktioner nära varandra i koden bidrar till skalbarheten på ett tydligt sätt.

Undvik kodduplicering då detta skapar många beroenden i koden som leder till försämrad skalbarhet. Det är bättre att ha så lite upprepning som möjligt.

Uppgift

I följande kapitel följer en beskrivning på hur programmet ska fungera och vilka operationer som ska fungera. Notera att alla steg som beskrivs *måste* följas för att bli godkänd på laborationen. Det går självklart bra att lägga till steg och funktionalitet utöver det som nämns här, men du måste *minst* ha med allting som tas upp här.

I slutet av kapitlet finns det ett antal körexempel som du kan titta på för att förstå hur programmet används.

Huvudprogrammet

För godkänt på den här laborationen krävs det att ditt program utför följande steg i samma ordning som de presenteras. Det är OK att lägga till mellansteg. Det är också helt OK att lägga till mer funktionalitet än det som anges här.

1. Öppna filen som anges i det första kommandoradsargumentet.
2. Läs in resterande kommandoradsargument till en annan behållare `arguments`. De första två kommandoradsargumenten (namnet på den körbarafilen och namnet på filen som öppnades i steg 1) ska inte finnas med i `arguments`, men resten ska vara med. Det är viktigt att ordningen på kommandoradsargumenten behålls. D.v.s. att när vi itererar `arguments` så ska sekvensen av argument vara samma som den sekvens användaren matade in.
3. Läs in alla ord från filen som öppnades i steg 1 till en lämplig behållare `text`. Ett ord i det här sammanhanget är en sekvens av icke-blanka tecken.
4. För varje argument `arg` i `arguments` ska följande göras:
    1. Hitta den första förekomsten av tecknet `=` i `arg` (om det finns).
    2. Dela upp `arg` i två delar:
        - `flag` vilket är allting som står till vänster om det hittade `=` tecknet. Om inget `=` tecken hittades ska den här strängen vara hela `arg`.
        - `parameter` vilket är allting som står till höger om det hittade `=` tecknet. Om inget `=` tecken hittades så ska den här strängen vara tom.

3. Avgör vilken operation som `flag` motsvarar och utför den operation (se nedan för en beskrivning av alla flaggor). Det är helt OK att göra en fullständig uppräkning (`if-else if` struktur) för det här.

Kom ihåg att det ska vara enkelt att lägga till nya flaggor och operationer. Observera att alla flaggor och operationer utförs *direkt* när de processeras. Detta betyder att det kan bli olika resultat beroende på i vilken ordning flaggorna kommer.

Operationer och flaggor

Ditt program ska ha stöd för följande flaggor. Observera att allting som står till höger om `=` i vissa av dessa flaggor är *parametrar*. Dessa extraherades i ett tidigare steg till variabeln `parameter`. Allting som är omslutet med `<` och `>` representerar inmatning från användaren. Större än och mindre än tecknen är inte med i användarens inmatning (se körexempel längre ner).

`--print` skriver ut alla ord i `text`, separerade med mellanslag, till `std::cout`.

`--frequency` Skriver ut en frekvenstabell (se nedan) som är sorterad i fallande ordning på antalet förekomster (alltså: vanligaste förekommande ordet först). Orden måste vara högerjusterade i utskriften av frekvenstabellen.

`--table` Skriver ut en frekvenstabell (se nedan) där orden är sorterade i bokstavsordning, A till Z. Orden måste vara vänsterjusterade i utskriften av frekvenstabellen.

`--substitute=<old>+<new>` byter ut alla förekomster av `<old>` i `text` med `<new>`, där `<old>` och `<new>` ersätts med godtyckliga strängar av användaren. För att extrahera orden `<old>` och `<new>` kan du dela upp variabeln `parameter` i två delar runt `+`, precis som när vi delar upp `arg` i delarna `flag` och `parameter`.

`--remove=<word>` tar bort alla hela förekomster av ordet `<word>` i `text`, där `<word>` är en godtyckligt sträng som anges av användaren. Notera här att du måste *ta bort alla förekomster* av ordet från `text`.

OBS: `--remove` och `--substitute` tar bort/byter endast ut *hela* förekomster av ord, aldrig delar av ett ord. T.ex. `--remove=the` kommer *inte* ta bort ordet `these` även fast `the` finns med i ordet. Utan det är endast när ordet matchar *exakt* som det ska tas bort.

Frekvenstabell

En central del av denna laboration är att konstruera och skriva ut så kallade *frekvenstabeller*. En frekvenstabell är en tabell som associerar varje unikt ord som förekommer i `text` med antalet gånger det förekommer i `text`. När en frekvenstabell skrivs ut så ska den skrivas ut på ett strukturerat format:

- Första kolumnen ska vara exakt tillräckligt bred så att det längsta ordet i `text` får plats. Detta betyder alltså att första kolumnen ska ha samma bredd för alla ord, oavsett hur långt ordet är. Om ordet är vänster- eller högerjusterat varierar beroende på vilken typ av frekvenstabell det handlar om (`--frequency` eller `--table`).

- Den andra kolumnen ska innehålla antalet förekomster av ordet som anges i den första kolumnen har i `text`. Detta ska alltså representeras av ett heltal.

Körexempel

```
$ ./a.out short.txt --print
Programming is fun Especially when you get to use the STL
which stands for the Standard Template Library which is the
name of the C++ standard library

$ ./a.out short.txt --remove=the --print
Programming is fun Especially when you get to use STL which
stands for Standard Template Library which is name of C++
standard library

$ ./a.out short.txt --substitute=the+WORD --print
Programming is fun Especially when you get to use WORD STL
which stands for WORD Standard Template Library which is
WORD name of WORD C++ standard library
```

Notera: Punkterna i de två nästkommande exempel är bara för att reducera storleken på utskriften i detta dokument. Det är ingenting som ska vara med i din utskrift.

```
$ ./a.out short.txt --frequency
        the 4
      which 2
         is 2
    library 1
    .
    .
    .
        STL 1
Programming 1
    Library 1
 Especially 1

$ ./a.out short.txt --table
C++         1
Especially  1
Library     1
Programming 1
    .
    .
    .
use         1
```

```
when        1
which       2
you         1
```

Man kan också kombinera flaggor:

```
$ ./a.out short.txt --substitute=the+THE --remove=C++ --print
Programming is fun Especially when you get to use THE STL
which stands for THE Standard Template Library which is THE
name of THE standard library
```

Krav

Som tidigare nämnt så handlar den här uppgiften om STL. Specifikt handlar det om algoritmer, databehållare och iteratorer. För att bli godkänd på den här labben måste du:

- Visa på att du kan följa en specifikation genom att implementera de steg som efterfrågas.
- Använda algoritmer på ett "rimligt" sätt. Detta för att visa på att du kan dela upp ett stort i problem i mindre delar som sedan kan lösas m.h.a. de algoritmer som C++ tillhandahåller.

  "Rimligt" i detta fall betyder att algoritmerna används för dess avsedda syfte: `std::sort` används för sortering, `std::min_element` används för att hitta det minsta element o.s.v.
- Demonstrera att du förstår användningsområdena för de olika databehållarna. Detta gör du genom att välja lämpliga databehållare för att spara data i ditt program.
- Skriva kod som är *korrekt*, *läsbar* och *skalbar* (se ovan för beskrivning av dessa).

Det är viktigt att du kan motivera dina val i uppgiften. Varför valde du de algoritmer du gjorde? Varför har du valt de databehållare som förekommer i din kod?

Hur börjar jag?

Du kan börja den här laborationen med att implementera huvudprogrammet. Följ stegen som anges för huvudprogrammet i uppgiften. Om du har svårt att sätta dig in i standardbiblioteket så kan du börja med att skriva huvudprogrammet med vanliga loopar och senare ersätta dem med lämpliga algoritmer.

Ett av målen med den här laborationen är att introducera ett nytt tankesätt. När man jobbar med standardbiblioteket bör man dela upp sitt program i många mindre problem och sedan fundera på huruvida det finns algoritmer som kan lösa dessa mindre problem åt oss. Det handlar mycket mer om att kombinera verktyg på ett effektivt sätt, snarare än att lösa problemet.

Tankesätt med algoritmer

Detta kapitel är frivilligt att läsa.
I det här kapitlet demonstreras det hur man kan tänka för att lösa problem med hjälp av standardbibliotekets algoritmer. Detta görs genom att identifiera lämpliga algoritmer för att lösa ett specifikt exempelproblem.

När man jobbar med algoritmer i STL så måste man först och främst identifiera *vilket* problem som ska lösas. I detta kapitel kommer vi lösa följande problem:

> Givet en vektor med heltal, hitta hur många av dessa heltal som är positiva efter vi har subtraherat det minsta talet två gånger.

För att lösa detta problem behöver vi först och främst identifiera vilka delproblem vi har. I det här fallet så behöver vi lösa följande problem:

- Hitta det minsta talet i en vektor
- Subtrahera ett givet tal från varje tal i en vektor
- Räkna förekomsten av positiva tal i en vektor

Nu är nästa steg att gå igenom listan av algoritmer (`https://en.cppreference.com/w/cpp/algorithm`) och se om det finns några som löser just våra problem.

För att göra detta kan det vara enklare för oss om vi identifierar vilka kategorier vi ska leta i. T.ex. att hitta det minsta talet i en vektor låter som att det skulle passa i någon av följande kategorier: *Non-modifying sequence operations* (att hitta det minsta talet kräver ingen modifikation av vektorn), *Sorting operations* (om man sorterar en vektor så blir det väldigt lätt att hitta det största talet) eller *Minimum/maximum operations* (vi vill hitta det minsta talet, vektorns *minimum*).

Att sortera vektorn låter som onödigt mycket jobb, så vi kan börja med att försöka hitta något som är enklare. Efter lite efterforskning hittar vi algoritmen `std::min_element` som verkar göra just det vi vill (den ligger i *Minimum/maximum operations*). Så med den algoritmen har vi löst det första problemet.

Nästa problem: att subtrahera ett givet tal från varje element i en vektor. Detta kommer kräva att vi på något sätt modifierar elementen i en vektor, så en bra start är att leta efter algoritmer i kategorin *Modifying sequence operations*.

Efter lite efterforskning hittar vi algoritmen `std::transform` som har beskrivningen:

> applies a function to a range of elements, storing results in a destination range.

Det beskriver inte exakt vad vi vill lösa, men det är det närmsta vi hittar. Så här måste vi även anpassa vårt problem efter algoritmen. Att subtrahera ett tal från ett annat tal är något en funktion kan lösa

och vi kan spara resultatet av den funktionen i samma vektor som vi läser ifrån. Så `std::transform` kan alltså användas här för att subtrahera det minsta talet två gånger.

Det sista problemet att lösa är att räkna antalet förekomster. Detta kräver ingen modifiering av vektorn så en bra start är att återigen kolla i kategorin *Non-modifying sequence operations*.

Där hittar vi följande algoritmer som potentiellt kan användas:

- `std::count`
- `std::count_if`
- `std::find_if`

`std::find_if` hittar endast det *första* element som uppfyller ett visst kriterium, så det är inte vad vi letar efter. `std::count` hittar antalet förekomster av ett visst *värde* så det är inte heller riktigt vad vi vill ha. `std::count_if` hittar antalet element som uppfyller ett *visst kriterium*.

Att ett tal ska vara *positivit* är ett kriterium, så `std::count_if` låter perfekt för att räkna antalet positiva tal.

Nu när vi har identifierat vilka algoritmer vi ska använda: `std::min_element`, `std::transform` och `std::count_if` så kvarstår det bara för oss att kombinera detta på ett lämpligt sätt för att lösa problemet, vilket lämnas som en övning till läsaren.