Linköpings universitet
Institutionen för datavetenskap (IDA)
UPP-gruppen 2018-05-11

Sorterad array

Mål

I en tidigare laboration implementerade ni en sorterad länkad lista som innehöll element av typen `int`. Den här laborationen går ut på att utöka en given array-klass som automatiskt sorterar datan. Användaren får specificera vilken datatyp arrayen innehåller och hur sorteringen ska gå till med hjälp av *C++ mallar* och *komparatorer*.
Målet är att utöka arrayen på ett sådant sätt att den kan användas mer generellt, i likhet med `std::vector`.

Läsanvisningar

- Funktionsmallar (16.1.1 i Primer)
  - deklaration
  - instansiering
- Klassmallar (16.1.2-16.1.3 i Primer)
  - deklaration
  - instansiering
  - definition av medlemsfunktioner
  - standardargument till template-parametrar

Redovisning

Den här labben innehåller deluppgifter 1, 2A, 2B och 2C. Dessa agerar endast som steg för att underlätta implementationen av laborationen. Varje deluppgift utökar den förgående deluppgiften med ny funktionalitet därför behöver ni endast redovisa slutresultatet i del 2C.

Uppgift 1: Mallifiering

Läs noga igenom den givna koden. För tillfället hanterar den sorterade arrayen endast element av datatypen `int`. I den här uppgiften ska ni generalisera arrayen på ett sådant sätt att användaren kan ange vilken datatyp som elementen är (exempelvis `std::string`, `double`, `int`, o.s.v.). Detta ska ni lösa med hjälp av C++ mallar (`template`).

Gör klassen `Sorted_Array` till en mall och gå sedan igenom hela arrayen (både deklarations- och implementationsfilen) för att lista ut vilka deklarationer som behöver ändras. Här är det viktigt att överväga huruvida implementationen av någon viss funktionalitet i arrayen måste ändras. Modifiera även det givna testprogrammet för att få det att fungera med den nu mallifierade varianten.

Det är viktigt att komma ihåg skillnaden mellan en mall och en klass. En klass kan genereras från en mall av kompilatorn; detta sker när en mall *instansieras* (d.v.s. när mallen blir tilldelad en datatyp). Innan en datatyp har tilldelats finns det ingen klass utan bara en mall.

Instansiering av mallar kan misslyckas om den angivna datatypen skulle leda till felaktig kod i definitionen av den genererade klassen, därför måste kompilatorn känna till hela definitionen av klassen när en instansiering sker.

Vad detta innebär rent praktiskt är att vid instansiering av mallar räcker det inte att kompilatorn känner till alla deklarationer utan den måste även känna till implementationen av hela mallen. Detta innebär att vi inte kan kompilera cc-filen för en mallifierad klass, utan vi måste istället se till att hela definitionen av klassen finns tillgänglig direkt i h-filen.

Detta löses lättast genom att inkludera cc-filen i h-filen. Fundera noga på huruvida inkluderingen ska ligga innanför eller utanför h-filens include-guards, diskutera gärna vad ni kom fram till med en assistent.

Denna deluppgift måste lösas med hjälp av inkludering, ni får inte flytta implementationen till h-filen.

Det finns ett antal givna testfall i `sorted_array_test.cc`. Lägg till fler testfall som testar arrayen för olika datatyper; minst `std::string` och `double`.

Ni kan gå vidare till nästa deluppgift när alla testfallen går igenom.

Uppgift 2: Sorteringsordning

För tillfället sorteras alltid arrayen i stigande ordning (minst till störst) med hjälp av `operator<`. Nu när ni har mallifierat klassen kan vi inte längre anta att `operator<` nödvändigtvis existerar för den angivna datatypen och även om den gör det, varför ska vi tvinga användaren av arrayen att alltid sortera sina värden i stigande ordning?

I många fall vill vi att sorteringen sker på något annat sätt, t.ex. i fallande ordning (störst till minst), eller kanske i lexikografisk ordning (bokstavsordning).

I den här uppgiften ska ni införa möjlighet för användaren att specificera hur sorteringen ska gå till genom att tillåta att en *komparator* anges vid instansieringen av mallen.

En komparator är en klass som innehåller en funktion `compare` som tar två element från arrayen som argument och returnerar `true` om det första elementet ska stå före det andra i sorteringsordningen och `false` annars. Denna klass används för att definiera en sorteringsordning av värden.

Tanken är att användaren själv ska kunna skapa en komparator för sin egen datatyp för att därmed kunna använda den i samband med vår sorterade array. Detta tillåter användaren att själv specificera vad som menas med att arrayen är sorterad.

Uppgift 2A: Stigande ordning

Den enklaste komparatorn är den som använder `operator<` för att jämföra om ett objekt är mindre än ett annat. Börja med att implementera en mallifierad klass `Less` vars `compare` funktion tar två värden av en godtycklig typ och returnerar `true` om det första argumentet är mindre än det andra och `false` annars. Notera att `compare` ska vara en icke-statisk medlemsfunktion i `Less`. Se till att testa komparatorn för olika datatyper.

Gå nu tillbaka till er sorterade array klass. Modifiera koden så att den använder `Less` istället för `operator<` vid insättning. För att detta ska fungera måste klassen innehålla en instans av komparatorn som vi kan anropa `compare` ifrån. Modifiera konstruktorerna i `Sorted_Array` så att de default-initierar komparatorn. Tänk på att komparatorn måste instansieras med samma typ som den sorterade arrayen.

Uppgift 2B: Sortera efter avstånd

Annledningen till att `compare` funktionen ska vara en icke-statisk medlemsfunktion är för att användaren ska kunna spara datamedlemmar i komparatorn om det behövs.

Ett fall när användaren måste spara data i komparatorn är om vår sorterade array ska vara sorterad efter avståndet från ett visst värde. Komparatorn har då tre värden att hålla koll på inuti `compare`; den första och den andra parametern, men även värdet som vi ska räkna ut avståndet till. Värdet vi ska räkna ut avståndet till anges av användaren när denne skapar sin komparator.

Exempel: säg att vår sorterade array innehåller element av datatypen `int`. Vi vill sortera arrayen så att alla tal som ligger nära värdet `10` hamnar i början av arrayen, medan tal som ligger längre ifrån `10` hamnar längre bak.

Om vi stoppar in följande värden:

```
5, 14, 0, -15, 19, 10, 100
```

och dessa element sorteras efter avståndet till `10` i stigande ordning (element närmare `10` ska vara längre fram i arrayen) så kommer arrayen se ut såhär:

```
10, 14, 5, 19, 0, -15, 100
```

Detta för att exempelvis avståndet från `19` till `10` är `9`, medan avståndet från `0` till `10` är `10` vilket är större än `9`, så `19` hamnar före `0` i den sorterade ordningen. Notera att avståndet aldrig ska vara negativt. Fallet när avstånden är lika behöver ni inte hantera, det löser sig automatiskt på grund av hur insättningen går till.

Nu ska ni implementera en komparatorklass vid namn `Distance` som har en datamedlem av en godtycklig datatyp (anges av `template`-parametern till `Distance`) vid namn `center`. Datamedlemmen `center` ska vara privat och ska initieras med hjälp av en konstruktor. Funktionen `compare` ska ta två parametrar och returnerar `true` om den första parametern har ett mindre avstånd till `center` än den andra parametern. Även den här klassen måste vara en mall.

Uppgift 2C: Godtycklig komparator i arrayen

För tillfället kan Distance inte användas i samband med er sorterade array. I den här deluppgiften ska ni modifiera er `Sorted_Array` klass så att den tar emot ytterligare en mallparameter. Den nya mallparametern ska representera vilken komparator som används för sorteringen. Lägg till en ny konstruktor för `Sorted_Array` som tar emot en instans av den specificerade komparatortypen och sparar den i klassen. Detta för att användaren ska kunna instansiera sin komparator innan den skickas in till arrayen.

Observera att `std::initializer_list` konstruktorn inte kommer fungera likadant som innan om ni lägger till en ny parameter. Säg att vi har `Sorted_Array(std::initalizer_list<T> a, double c)` då måste ni anropa konstruktorn med en nästlad *brace-enclosed-list* såhär:

```
Sorted_Array a {{1,2,3,4},5.7};
```

Nu kan arrayen används väldigt generellt, men det kräver också mer av användaren eftersom denne måste implementera en komparator för sin datatyp. Detta kan vara ett hinder om användaren bara vill ha en `int`-array som är sorterad i stigande ordning. Därför ska ni lägga till ett default-argument för komparatortypen i `template`-argumentlistan så att om användaren inte specificerar den sista `template`-parametern kommer vi automatiskt att välja `Less<T>`, där `T` är den första `template`-parametern.

Testfall

I den givna koden finns det en mängd testfall. Se till att utöka dessa så att ni testar för flera olika datatyper och för både `Distance` och `Less`. Testa även att default-argumentet fungerar som det ska genom att i något testfall inte ange den andra `template`-parametern. Det är inte krav på att ni använder specifikt catch för dessa tester, men det är starkt rekommenderat.