LINKÖPINGS TEKNISKA HÖGSKOLA
Institutionen för datavetenskap
Programvara och system
Tommy Olsson

2015-03-06

TDDI14 Objektorienterad programmering

Laboration Listan

Introduktion

Laborationen ska kunna göras efter föreläsning 2 (klasser och operatoröverlagring) och med kunskaper från kursen TDIU04, speciellt namnrymder (**namespace**) och mallar (**template**). Ett antal viktiga tillägg ska göras i en given contrainerklass List. List har påtagliga brister i funktionalitet och är inte säker att använda, av olika anledningar. Dessutom kan endast heltalsvärden lagras. Du ska modifiera List så att den blir funktionell, säker och generell. Given kod: http://www.ida.liu.se/~TDDI14/lab/.

List är given i två versioner, en med *iterativa* funktioner (det normala för en enkellänkad lista), en med *rekursiva* funktioner. Välj den ena att arbeta med och var konsekvent med avseende på iteration eller rekursion för egna funktioner.

Modifiera List_Node

Lägg till följande:

- en rekursiv destruktor, så att destruktorn för List fungerar som tänkt, dvs att det i den räcker med ’**delete** head_’ för att alla noder i listan ska destrueras.
- lägg till konstruktor(er) för att skapa List_Node-objekt på följande sätt:

```
list = new List_Node(value, tail);   // tail pekar på en List_Node eller är nullptr
list = new List_Node(value);         // next_ sätts till nullptr
```

- defaultkonstruktor, kopieringskonstruktor och kopieringstilldelningsoperator ska *inte* finnas och *inte* heller move-konstruktor eller move-tilldelningsoperator.

Lägg till List-operationer

Lägg till följande i List:

- en privat medlemsfunktion **copy**() som ska göra en kopia av en lista av List_Noder; en pekare till listan som ska kopieras ska ges som argument, en pekare till kopian ska returneras
- en medlemsfunktion **swap**() som byter innehåll med en annan lista
- en icke-medlemsfunktion swap() som byter innehåll på två listor
- en *kopieringskonstruktor* som gör djup kopiering
- en *kopieringstilldelningsoperator* som gör djup kopiering, implementerad med idiomet ”skapa en temporär och byt”
- en *move-konstruktor*, som flyttar innehållet från källobjektet till destinationsobjektet och ”nollställer” källobjektet på ett sätt som gör att det är säkert att fortsätta använda.

  Testa move-konstruktorn och även move-tilldelningsoperatorn (se nedan) med hjälpfunktionen std::move() (<utility>).

```
List lista_1;
...
List lista_2(lista_1);            // kopieringskonstruktorn används
List lista_3(std::move(lista_1)); // move-konstruktorn används (om den finns)
```

- en *move-tilldelningsoperator*, som ska flytta innehållet från högeroperanden till vänsteroperanden. Eftersom List är en containerklass bör detta innebära att högeroperanden är en tom lista efter detta.

```
lista_1 = std::move(lista_2);
```

- en konstruktor som kan initiera en lista med värden från en *initierarlista*, std::initializer_list.

```
List lista{ 1, 2, 3, 4, 5 };
```

  I en initierarlista är elementen tillgängliga via iteratorer som erhålls på vanligt sätt med medlemsfunktionerna begin() och end(). Det finns även en funktion size() om man behöver ta reda på hur många värden som finns i en initieralista.

Se till att alla operationer ovan testas noga av programmet!

Kontrollera minneshanteringen
Gå igenom koden och kontrollera där dynamiskt minnestilldelning görs (**new**) och se till att inget dynamiskt minne förloras om undantag kastas av någon anledning. Om exempelvis copy() blir avbruten efter att endast en del av listan blivit kopierad måste den redan kopierade delen återlämnas.

Inför en egen namnrymd för List
Kapsla List och dess operationer i en egen *namnrymd* med namnet linked_list och modifiera test-programmet för detta.

Gör om List till mall
Gör om List till en *klassmall* (**template**) så att den typ av element som ska lagras i en lista kan väljas.

List.cc döps lämpligtvis om till List.tcc (’t’ som i template) och inkluderas i slutet av List.h. Modifiera list-test.cc och Makefile. Alla separatdefinierade medlemmar av en klassmall måste ha samma mall-parameterlista som klassmallen de tillhör.

Lägg till iteratorer i List
List ska ha samma funktionalitet då det gäller iteratorer som standardbibliotekets containrar:

- typmedlemmarna iterator och const_iterator (implementering beskrivs nedan)
- medlemsfunktionerna begin() och cbegin() för att returnera en iterator till det första elementet i en lista eller en förbi-sista-iterator om listan är tom. begin() ska finnas i två versioner, icke-**const** som returnerar iterator, **const** som returnerar const_iterator. const-versionen väljs automatiskt om list-objektet är **const**. cbegin() ska returnera const_iterator, oavsett om listobjektet är **const** eller inte.
- medlemsfunktionerna end() och cend() för att returnera en förbi-sista-iterator. För övrigt analogt med vad som sagts för begin() och cbegin() ovan.

Implementeringsmässigt ska en List-iterator att vara en pekare till en listnod. När man applicerar **operator*** eller **operator->** är det *värdet* som lagras i listnoden som man ska få åtkomst till via en referens respektive en pekare. Observera, semantiken för en egen **operator->** är speciell, den inbyggda **operator->** kommer underförstått att appliceras på den pekare som den egna **operator->** returnerar.

Använd kodskelettet nedan som utgångspunkt för att definiera List_iterator_ (implementering för List::iterator) och List_const_iterator_ (implementering för List::const_iterator):

```
template<typename T>
struct List_iterator_
{
   using value_type = T;
   using pointer = T*;
   using reference = T&;
   using difference_type = std::ptrdiff_t;
   using iterator_category = std::forward_iterator_tag;

   // defaultkonstruktor som sätter iteratorn till "förbi-sista"
   // konstruktor för att initiera med en pekare till en listnod
   // operator* ska returnera en referens till elementet i noden
   // operator-> ska returnera en pekare till elementet i noden
   // operator++ i både prefix- och postfix-version för att stega iteratorn
   // operator== för att kontrollera om två iteratorer är lika, eller inte
   // operator!= för att kontrollera om två iteratorer är olika, eller inte
   // pekare till listnod, representationen för iteratorn
};
```

Det avslutande understrykningstecknet i List_iterator_ markerar att klassen tillhör implementeringen och är inget som användaren behöver känna till, även om den följer med i List.h. Fem grundläggande typer definieras som medlemmar (exempelvis kräver vissa algoritmer dessa; dessa typeer kan även ärvas från hjälpklassen std::iterator):

- *value_type* är ett annat namn för elementtypen, T
- *pointer* är typen för pekare till elementtypen, typiskt T*, **const** T* för const_iterator
- *reference* är typen för referens till elementtypen, typiskt T&, **const** T* för const_iterator

- *difference_type* anger resultattypen ifall man bildar differensen mellan två iteratorer (inte aktuellt för forward-iteratorer), typiskt används std::ptrdiff_t
- *iterator_category* anger vilken kategori av iterator de gäller – input, output, forward, bidirectional eller random access. I vårt fall är det forward iterator som gäller eftersom det är vad en enkellänkad lista tillåter – stega framåt ett element i taget och kunna läsa och skriva (om inte const_iterator).

Låt alla medlemmar vara **public**, även pekaren. *Definiera* alla medlemsfunktioner i klassen, gör inga separata definitioner.

Definiera List_iterator_ och List_const_iterator_ efter definitionen av List_Node men före List. Definiera sedan iterator och const_iterator som medlemmar i List, som synonym för List_iterator_<T> respektive List_const_iterator_<T> (se nedan). I C++ direkt finns flera exempel där iteratorer definieras inuti den klass de tillhör, gör inte det.

Iteratoroperationer (det är *mycket viktigt* att använda **const** korrekt vid deklaration av medlems-funktioner och parametrar!):

- defaultkonstruktor som initierar en iterator till att vare en förbi-sista-iterator (nullptr)
- konstruktor som initierar en iterator till att peka på den första noden i en lista (head_) eller till en förbi-sista-iterator om listan är tom
- const_iterator ska ha en konstruktor som omvandlar från iterator (List_iterator_<T>) till const_iterator (List_const_iterator_<T>)
- kopieringskonstruktor och kopieringstilldelningsoperator ska finnas men *inte* flyttkonstruktor och flyttilldelningsoperator
- **operator*** ska returnera en referens till elementet (data_) i listnoden; en const_iterator ska returnera referens till konstant
- **operator->** ska returnera en pekare (adress) till elementet (data_) i listnoden; en const_iterator ska returnera pekare till konstant
- **operator++** ska stega fram till nästa nod i listan (next_) – både prefix- och postfix-version ska finnas, med normal semantik med avseende på returvärdet
- **operator==** ska jämföra två iteratorer med avseende på likhet (eller inte)
- **operator!=** ska jämföra två iteratorer med avseende på olikhet (eller inte)
- **operator==** och **operator!=** ska även kunna jämföra en iterator (List_iterator_<T>) med en const_iterator (List_const_iterator_<T>) och det kräver även en icke-medlemsversion av respektive operator, som komplement till medlemsversionen

Tillägg i List (definiera funktionerna i List, definiera inte separat):

- definiera iterator som synonym för List_iterator_<T> och const_iterator som synonym för List_const_iterator_<T>; **typedef** eller alias-deklarationer (**using**)
- begin() ska returnera en iterator som pekar på det första elementet i en lista eller förbi-sista-iterator om listan är tom; om **const** List ska const_iterator returneras (funktionen måste alltså överlagras i två versioner)
- end() ska returnera en iterator som är en förbi-sista-iterator; om **const** List ska const_iterator returneras
- cbegin() och cend() ska returnera const_iterator, oavsett om listan ifråga är **const** eller inte

Arbeta stegvis. Börja med iterator och funktionerna begin() och end(). När det fungerar kan du till implementera alla varianter begin()/end()/cbegin()/cend() och sedan const_iterator.

Ta bort medlemsfunktionen print() och ersätt utskrifterna i testprogrammet med något iteratorbaserat, till exempel olika varianter av **for**-satser (vanlig och intervallbaserad) eller algoritmer, som exempelvis

```
copy(list.cbegin(), list.cend(), ostream_iterator<int>(cout, " "));
```

Testa List noga med avseeden på iteratorer och allt annat innan du redovisar!

List.h, List.tcc, det modifierade testprogram på list-test.cc (test efter mallifieringen) och separat test-program för iteratorerna på filen list-iterator-test.cc ska skickas in.