Linköpings Universitet
Institutionen för datavetenskap (IDA)
UPP-gruppen 2025-02-25

Smartpekare

Introduktion

I standardbiblioteket för C++ finns det s.k. *smartpekare* som underlättar minneshantering avsevärt genom att sköta avallokering av minne automatiskt. Det finns specifikt två smartpekare i STL:

- `std::unique_ptr<T>` som lagrar ett dynamiskt allokerat `T`-objekt. Denna variant förbjuder användaren från att kopiera pekaren. Istället måste man använda *flyttsemantik* för att flytta runt pekaren. En flytt innebär att föregående plats blir `nullptr` och den nya platsen pekar nu istället på objektet. När pekaren destrueras så avallokeras objektet. Det är alltså endast tillåtet att ha *en* `std::unique_ptr<T>` till varje `T`-objekt. Denna begränsning gör att pekaren garanterar att objektet avallokeras när det inte längre används och att det inte kan uppstå flera pekare till samma objekt.
- `std::shared_ptr<T>` lagrar ett dynamiskt allokerat `T`-objekt. Till skillnad från den tidigare beskrvina smartpekaren `std::unique_ptr<T>` så tillåter `std::shared_ptr<T>` att man har flera pekare till samma objekt. Objektet som pekas på avallokeras endast när det inte längre finns några `std::shared_ptr<T>` som pekar på det. Denna pekare går att kopiera vilket leder till att ansvaret för att avallokera objektet hamnar på den sista existerande pekaren till objektet.

Dessa smartpekare är inte en direkt ersättning för vanliga pekare, utan de är verktyg för att underlätta avallokering av minne. Framförallt är de komponenter som begränsar vad som är tillåtet att göra med dem, vilket leder till att de har starkare garantier för hur minnet kan användas.

De smartpekarna kommer med sin egna uppsättning problem och brister som man måste ta hänsyn till. Men när man behärskar dessa kan man enkelt rikta om sin tankekraft till andra problem än just minneshantering.

Därför ska man *inte* tänka på smartpekare som en ersättare för det vi har lärt oss i tidigare kurser om minne, utan det är ett komplement, en vidareutveckling på hur minne kan hanteras. De är inte heller en universallösning för minne. Smartpekare introducerar strikta begränsningar som inte är lämpliga för alla situationer.

Mål

I denna laborationsuppgift ska du utforska den senare varianten av smartpekare, `std::shared_ptr<T>` genom att implementera din egna variant vid namn `counted_ptr<T>`. För att göra detta kommer du använda *mallar*, *speciella medlemsfunktioner*, *operatoröverlagring* och *minneshantering*.

Sedan kommer du implementera ett enkelt scenario där `counted_ptr<T>` används för att förenkla minneshanteringen, samt för att öva på att använda smartpekare.

I denna laboration är det viktigt att du fokuserar på *generaliserbarhet* samt *användbarhet* av koden som produceras.

Generaliserbarhet

Mallar används i regel för att skriva generellt tillämpningsbara moduler av kod som kan användas för flera olika syften. Detta innebär rent konkret att koden du skriver inte ska vara bunden till ett specifikt användningsområde utan det ska på ett tydligt sätt endast knyta an till det problem som den generella modulen behandlar.

I detta fall innebär det att `counted_ptr<T>` endast ska representera en pekare som håller koll på hur många som pekar till den platsen. Användaren ska alltså inte vara bunden till ett specifikt användningsområde (t.ex. att pekaren kan användas för att hålla koll på spelobjekt, eller att pekaren används för att implementera en länkad lista etc.). Därför ska man välja namn på variabler, datamedlemmar och funktioner som uttrycker exakt det de är utan att ge några specifika indikationer på vad man *kan* använda klassen för.

En annan aspekt av detta är att datatypen som pekaren pekar på ska ha så få begränsningar som möjligt. Man vill att det ska fungera för så många datatyper som möjligt. Varje operation du utför på en generell datatyp `T` medför någon typ av begränsning. Om du t.ex. skapar en variabel av datatypen `T` måste denna variabel initieras på något vis. Hur du väljer att initiera variabeln sätter begränsningar på hur du *antar* att datatypen kan användas. T.ex. kan du studera detta kodstycke:

```
T x { };
x += 5;
```

Fundera på vilka antaganden som faktiskt görs här. Finns det några datatyper som `T` *inte* kan vara här?

I samma anda kan man även fundera på hur parameterövering ska fungera för mallar. Eftersom att `T` är en *godtycklig* datatyp vet du alltså inte hur stort ett objekt är, och därför är det lämpligt att anpassa sig efter det *värsta* fallet. Alltså är det troligen klokt att använda konstanta referenser för det minimerar hur mycket som måste kopieras *i det allmänna fallet*.

Användbarhet

`counted_ptr<T>` är ett verktyg för abstraktion. I sig själv ger den inte användaren (programmeraren) någon ny funktionalitet, utan det handlar snarare om att underlätta för användaren att uppnå den funktionalitet de vill utveckla. Därför är det av yttersta vikt att användaren kan använda den smartpekaren utan att behöva lära sig ett nytt komplicerat gränssnitt. Det ska fungera i enlighet med det användaren redan behärskar.

I detta fall är abstraktionen i fråga något som förenklar minneshantering. Därför är det lämpligt att `counted_ptr<T>` efterliknar en ”vanlig” pekare i den mån som det är möjligt. Undvik därför att skapa en massa medlemsfunktioner eller andra konstruktioner som avviker från hur användaren vanligtvis skulle använda pekare.

Motivation

Denna laborationsuppgift tjänar flera syften för ditt lärande samt för kursens mål. Det första syftet är att introducera generisk programmering med hjälp av mallar. Denna uppgift tillåter dig att öva på din behärskning av hur mallar kan användas för att skriva generellt tillämpningsbara klasser och funktioner som kan användas vid flera olika situationer.

Det andra syftet är att ge inblick i hur du kan nyttja programmeringskonstruktioner för att abstrahera bort mycket manuellt arbete gällande minneshanteringen och allmän resurshantering. D.v.s. hur kan du se till att minne hanteras korrekt utan att du konstant behöver ha det i åtanke?

Det tredje syftet relaterar till det projekt som genomförs i kursen. I projekten som ska implementeras är målsättningen att lägga fokus på att konstruera en bra objektorienterad design och en stor del av detta är att hantera minnet av diverse olika dynamiskt allokerade objekt. Användningsområdena som uppstår inom projektet är mångfaldig och kan leda till mycket komplexitet och genom att nyttja de abstraktioner som introduceras i denna uppgift kan du reducera mycket av komplexiteten som uppstår kring just hantering av dynamiskt minne.

Därför uppmuntras du till att använda `std::shared_ptr<T>` där det är lämpligt så att du kan fokusera på funktionalitet snarare än minneshantering. Notera alltså att idéerna som presenteras i denna laboration är direkt tillämpningsbara på `std::shared_ptr<T>`.

Redovisning

Denna laborationsuppgift ska, precis som förra labben, demonstreras muntligt innan den kan skickas in för bedömning. Uppgiften består av tre delar, uppgift 1 & 2 som innebär att du skriver kod och uppgift 3 som behandlar teori. Du ska **inte** redovisa varje deluppgift för sig, utan du redovisar allting på en gång.

Hur fungerar `counted_ptr<T>`?

`counted_ptr<T>` representerar en pekare som använder *automatisk referensräkning*.

Detta innebär att en `counted_ptr<T>` håller koll på vilket objekt i minnet den pekar på, men också hur många *andra* `counted_ptr<T>` det finns som pekar på samma objekt.

Alltså måste varje `counted_ptr<T>` hålla koll på två saker:

1. Adressen till datan som för tillfället pekas på.
2. En räknare som håller koll på antalet pekare till minnet som pekas på.

Notera att räknaren *alltid* ska ha samma värde för alla pekare som pekar på samma objekt. Detta innebär alltså att räknaren måste vara delad mellan flera pekare (så den kan inte lagras direkt i `counted_ptr` objektet).

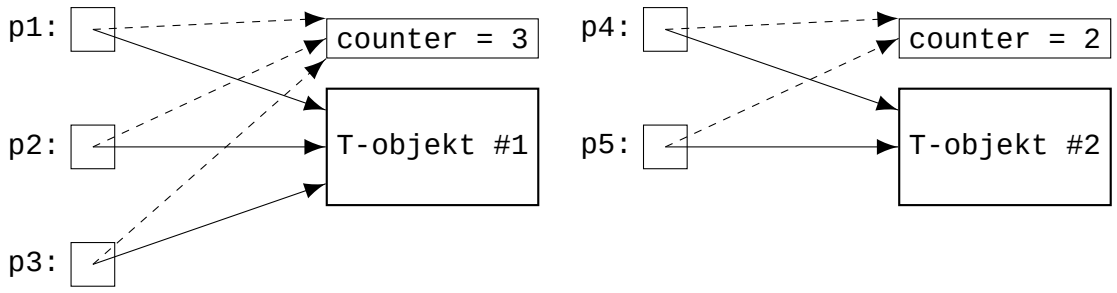
Varje gång en `counted_ptr<T>` kopieras ska den delade räknaren öka med ett. När `counted_ptr<T>` skapas ska räknaren börja på `1`.

Varje gång en `counted_ptr<T>` förstörs ska den delade räknare minska med ett. När räknaren når värdet noll, då avallokerar du objektet som pekas på (samt räknaren), eftersom det inte längre finns någon `counted_ptr<T>` som pekar på objektet.

På det viset garanterar du att räknaren alltid korrekt reflekterar mängden delade pekare.

Notera: räknaren är **inte** delad mellan *alla* `counted_ptr<T>` objekt, utan den är delad mellan alla `counted_ptr<T>` objekt som pekar på *samma* adress.

Exempel: Antag att du har två dynamiskt allokerade objekt av typen `T` och att du har fem stycken `counted_ptr<T>` vid namn `p1`, `p2`, `p3`, `p4` och `p5`. Pekarstrukturen ser om som nedan:



I detta diagram representerar de heldragna pilarna vilket objekt som den smartpekaren pekar på, medan de streckade pilarna visar vilken räknare som är aktuell för den smartpekaren.

I diagrammet kan man då se att pekarna `p1`, `p2` och `p3` pekar alla på `T-objekt #1` och att de är tre till antal. Därför är den delade räknaren satt till `3`.

Medan pekarna `p4` och `p5` pekar på `T-objekt #2`, vilket betyder att de har en delad räknare som är satt till `2`.

Uppgift 1: Implementera smartpekaren

Första delen av denna uppgift är att implementera `counted_ptr<T>` som en klassmall, där `T` är datatypen av objektet som din smartpekare pekar på. Det är viktigt att du förstår idéen av hur `counted_ptr<T>` fungerar innan du börjar skriva kod. Tanken är att användaren *dynamiskt allokerar* ett `T`-objekt och sedan lämnar över det till en `counted_ptr<T>` utan att behöva tänka på att det finns en underliggande räknare. Följande krav måste uppfyllas av din implementation av `counted_ptr`:

- Det ska finnas en `h`-fil som innehåller alla deklarationer och en `tcc`-fil som innehåller alla definitioner. Egenskrivna tester läggs i `uppgift1.cc`.
- Du måste skapa en konstruktor som tar emot en *vanlig* `T`-pekare och skapar en *ny* räknaren för objektet. Du kan anta att användaren aldrig skickar *samma* pekare till två olika `counted_ptr`.
- En standardkonstruktor (d.v.s. en konstruktor utan parametrar) ska implementeras. Denna konstruktor initierar smartpekaren till ett tillstånd som motsvarar `nullptr`. D.v.s. att den sätter ett tillstånd på `counted_ptr` som motsvarar att den inte pekar på någonting alls. Inget minne får allokeras i denna konstruktor.
- Det ska finnas både en kopieringskonstruktor och en kopieringstilldelningsoperator. Kom ihåg att under kopering ska du öka räknaren för det allokerade objektet. Vid tilldelning måste du också minska räkaren för pekaren som blir ersatt (objektet avallokeras om räknaren når noll).
- Det ska vara möjligt att flytta pekaren med hjälp av både en flyttkonstruktor samt en flyttilldelningsoperator. Notera att pekaren som flyttas ska motsvara `nullptr` när flytten är klar. Fundera på huruvida räknaren ska uppdateras eller ej.
- Det ska också vara möjligt för användaren att tilldela `nullptr` (men **inte** vanliga pekare) till smartpekaren, vilket minskar räknaren (och eventuellt avallokerar datan) och sätter smartpekaren i standardtillståndet. **Tips:** literalen `nullptr` är av datatypen `std::nullptr_t`.
- `operator*()` (returnerar en referens till objektet som pekas på) och `operator->()` (returnerar en vanlig pekare till objekt som pekas på) måste implementeras. Du behöver inte hantera fallet om smartpekare motsvarar `nullptr`.
- Det ska gå att jämföra två `counted_ptr` objekt m.h.a. `==` och `!=`. Jämförelsen görs på addressen av objektet som pekas på. Det ska också gå att jämföra smartpekarna med vanliga pekare.
- Det ska finnas en medlemsfunktion `use_count()` som returnerar `0` om smartpekaren motsvarar `nullptr`, och värdet av räknaren i vanliga fallet.
- En medlemsfunktion `get()` som returnerar en *vanlig* pekare till objektet som den smartpekaren för tillfället pekar på. Notera att denna pekare *inte* ökar räknaren.

Dessa krav ska fungera för `counted_ptr` objekt som både är `const` och icke-`const`. Om en `counted_ptr` är `const` ska det **inte** vara möjligt för användaren att modifiera objektet som pekas på.

Tips: Du bör skriva egna tester som testar varje funktion. Hur dessa tester görs är upp till dig. Men förslagsvis skriver du ett testprogram som använder `counted_ptr` på olika sätt.

Uppgift 2: Använd smartpekaren

I denna deluppgift ska du nu använda din `counted_ptr` och dina nyfunna färdigheter med mallar för att förbättra de givna filerna `node.h` och `node.cc`. Dessa filer innehåller en implementation av en struktur som kallas en *riktad graf* (engelska: *directed graph*). Denna struktur är lik en länkad lista, men har vissa viktiga skillnader:

- Varje nod har godtyckligt många länkar
- Flera olika noder kan länka till samma nod

För tillfället finns det två stora problem i implementation:

1. Strukturen leder till minnesläckor eftersom att inga noder någonsin avallokeras
2. Noderna är begränsade till att endast ha heltal som värden

Dessa ska du bemöta genom att lösa dessa delar:

(a) Skriv om klassen `Node` i `node.h` och `node.cc` så att den använder `counted_ptr` istället för vanliga pekare. Anpassa också det givna programmet i `uppgift2.cc` så att den fungerar med `counted_ptr`.

Notera att det finns ställen i koden där du fortfarande måste använda vanliga pekare, så fundera noggrannt på vart du kan använda `counted_ptr`.

För tillfället ger programmet skrivet i `uppgift2.cc` minnesläckor, det kan du se genom att köra `valgrind`:

```
$ valgrind ./a.out
```

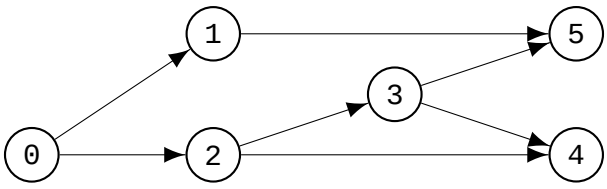
Efter omskrivning med `counted_ptr` ska dessa minnesläckor vara borta.

(b) Mallifiera `Node` så att den sparar värden av datatypen `T` som användaren själv kan välja. En del av detta kommer också innebära du måste göra om `node.cc` till en `node.tcc` fil.

Ändra även `uppgift2.cc` så att noderna innehåller strängar istället för heltal.

Tips: När du gör om filstrukturen till att använda `node.tcc` istället för `node.cc`, se till att `node.cc` inte längre finns kvar, detta minskar förvirringen för dig själv.

I `uppgift2.cc` skapas en riktad graf vars grafiska representation skulle kunna se ut såhär:



Där varje cirkel representerar en graf (med dess lagrade värde skrivet inuti cirkeln) och pilarna representerar länkar. Notera att en nod kan ha en varierad mängd länkar. Studera koden i `uppgift2.cc` noggrannt, och jämför med diagrammet ovan.

Frivilligt: Visualisering och experiment

För att hjälpa dig med din förståelse av denna del så finns det ett givet program som tillåter dig att modifiera och visualisera riktade grafer. Detta program kan du kompilera och köra genom att, från katalogen där filen `Makefile` ligger, köra kommandot:

```
$ make graph
```

Notera att källkoden för detta program finns tillgängligt i katalogen `lib`. Du ska aldrig behöva läsa eller ändra den koden. Om du får kompileringsfel när du kör `make graph` så beror det på att något är fel i noderna. Om du misstänker att felet faktiskt finns i den givna koden, prata med en assistent innan du börjar försöka göra ändringar.

Programmet använder primärt terminalen, men öppnar också ett fönster på sidan av där en visuell representation av grafen syns, det kan vara bra att se till att både fönstret och terminalen är synliga samtidigt. Programmet tillåter dig att lägga till och ta bort noder samt länkar från den riktade grafen. Detta tillåter dig att experimentera med själva strukturen.

Uppgift 3: Teorifrågor

I denna laborationsuppgift måste du besvara följande frågor i en textfil vid namn `answers.txt`, som du skickar in tillsammans med din kod efter godkänd redovisning.

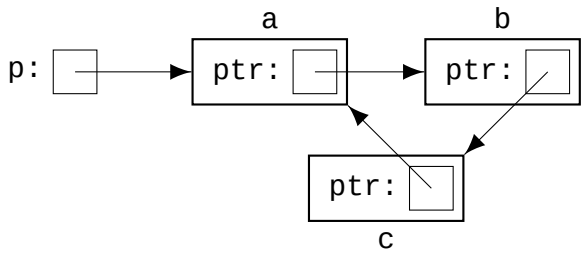
Krav: Ditt svar på *båda* frågorna ska vara mellan 100 till 200 ord långt.

1. I följande exempel används `Node` klassen från uppgift 2. Förklara varför for-loopen i exemplet nedan fungerar:

```
Node* n0 { new Node { 0 } };
Node* n1 { new Node { 1 } };
Node* n2 { new Node { 2 } };
n0->insert(n1);
n0->insert(n2);
for (Node* node : (*n0))
{
  std::cout << node->value << " ";
}
std::cout << std::endl;
```

   Vad gör for-loopen? Vad är det för något i `Node` klassen som gör att denna loop är möjlig?

2. Antag att du har följande struktur:



   Där varje pil representerar en `counted_ptr`. Varför blir det minnesläckor när du tar bort pekaren vid namn `p`? D.v.s. varför avallokeras inte hela strukturen? Notera att inga andra instanser av `counted_ptr` förekommer i programmet.

   Tips: Du kan testa ett liknande exempel i visualiseringen genom att köra:

```
$ make
$ build/graph insert node node node link 0 1 link 1 2 link 2 0 done
```

   Du kan också testa följande, vilket skapar samma graf och sedan avslutar direkt:

```
$ make
$ valgrind ./build/graph i n n n l 0 1 l 1 2 l 2 0 d q
```

   För att se att minnesläckor faktiskt sker i fallet beskrivet ovan! **Varning:** Det kan ta ett litet tag att köra detta, vänta tills programmet slutar av sig själv.