LINKÖPINGS TEKNISKA HÖGSKOLA
Institutionen för datavetenskap
Programvara och system
Tommy Olsson

2014-02-05

TDDI14 Objektorienterad programmering

Lektionsuppgifter

Lektionsplanering 2

Lektion 1 3

Lektion 2–3 5

Lektion 4 7

Lektionsplanering

Lektion 1

På föreläsning 1-2 har konstruktion av en enskild, icke-trivial klass String behandlats, speciellt avseende initiering, destruering, kopiering, tilldelning, iteratorer och move-semantik samt typomvandling och operatoröverlagring. Mallar (**template**) och namnrymder är känt från TDIU04, liksom egenskaper hos containrar och containeriteratorer.

Lektionen ska ge underlag för laboration Listan samt övning på design av en icke-trivial klass.

- genomgång av laboration Listan (namnrymder och enkla klassmallar är känt från TDIU04)
- övning på klass (initiering, destruering, kopiering, tilldelning och operatoröverlagring för en enkel smartpekartyp)

Lektion 2

På föreläsning 3-4 har härledda klasser, arv och polymorfi behandlats.

- genomgång av enkel användning av make
- övning på arv och polymorfi (namn–dubbelnamn), görs delvis på lektion 2, delvis på lektion 3

Lektion 3

På föreläsning 5 har bland annat uttryckträd behandlats.

Lektionen ska ge introduktion till laboration Kalylatorn samt vidare övning på arv och polymorfi (forts. på övningen namn–dubbelnamn)

- genomgång av laboration Kalkylatorn (utom klassen Variable_Table)
- övning på arv, polymorfi (forts. namn - dubbelnamn)

Lektion 4

På föreläsning 6 har mallar behandlats.

- övning på enkel klassmall (Wrapper), överlagring av **operator**<< och **operator**>> för Wrapper

Övningsuppgift för lektion 1

Uppgiften tar speciellt upp initiering, kopiering, tilldelning och destruering av klassobjekt samt operatoröverlagring. Uppgiften är allmänt förberedande för laborationsuppgifterna.

Antag att vi i ett program ska hantera dynamiskt minnestilldelade objekt av typen Integer:

```
class Integer
{
  public:
    explicit Integer(int value = 0) : value_(value) {}

    void set_value(int value) { value_ = value; }
    int  get_value() const { return value_; }

  private:
    int value_;
};
```

Vi vill säkerställa att minnet för sådana objekt alltid återlämnas då pekare som refererar till objekten upphör att existera. Om vi t.ex. deklarerar en pekare p i ett block

```
{
  Integer* p = new Integer(1);
  ...
}
```

och det inte görs **delete** på p innan blocket avslutas, kommer inte minnet för objektet som skapades med **new** att återlämnas när pekaren försvinner (minnesläcka uppstår). Ett sätt att lösa detta är att använda en ”smart pekare”. En smart pekare är ett klassobjekt som kapslar en vanlig, ”rå pekare” och vars destruktor ser till att minnet för objektet som pekaren pekar på återlämnas. Det är önskvärt att en smart pekare kan användas som en vanlig pekare, t.ex. att man kan applicera **operator*** och **operator->**.

```
{
  smart_pointer sp(new Integer(1));

  cout << sp->get_value() << endl;
  sp->set_value(2);
  (*sp).set_value(3);
  ...
} // minnet för objektet som skapades ovan återlämnas av ~smart_pointer()
```

Det kan också vara praktiskt att ha en medlemsfunktion swap för att byta de råa pekarna för två smarta pekare, och en medlemsfunktion get som returnera den råa pekaren för en smart pekare.

Det finns olika modeller för smarta pekare. I standardbiblioteket finns unique_ptr, som gör ”destruktiv kopiering” och ”destruktiv tilldelning”. Det innebär att kopieringskonstruktorn och kopierings-tilldelningsoperatorn är eliminerade men move-konstruktor och move-tilldelningsoperator finns och flyttar över den ”råa” pekaren från källan till mottagaren och sätter pekaren i källan till **nullptr**.

```
unique_ptr<Integer> ap1(new Integer(1));// smart pekare till Integer
unique_ptr<Integer> ap2(ap1);           // ap2 tar objektet från ap1
ap1 = ap2;                              // ap1 tar tillbaka objektet
```

”unique” syftar på att ägarskapet är ”unikt”, vilket avser att en pekare till ett objekt endast kan ägas av en unique_ptr i taget (om man använder unique_ptr korrekt).

Andra vanliga modeller för smarta pekare är:

- ”copy on construct/assign” eller ”deep copy” – då en sådan smart pekare kopieras eller tilldelas görs en kopia av objekt som pekaren i källan pekar på.
- ”copy on write” – då en sådan smart pekare kopieras eller tilldelas kopieras bara pekaren, dvs mottagaren och källans pekare pekar på samma objekt. Först om en operation utförs på smart pekare som ändrar på objektet görs en kopia av objektet för den smarta pekaren. Detta kräver att man också håller reda på när den sista pekaren till ett objekt försvinner, så att man först då återlämnar minnet för objektet (en sådan teknik är referensräkning).

Att konstruera smarta pekare som beter sig precis som vanliga pekare är inte enkelt och kräver att man använder flera avancerade konstruktioner i C++ och en inte helt enkel implementering. Här ska vi nöja oss med att göra en någorlunda enkel smart pekare, där syftet främst är att ta upp initiering, kopiering, tilldelning och destruering för klassobjekt samt operatoröverlagring.

Uppgiften

Konstruera en smart pekare kallad copied_pointer för att hantera objekt av typen Integer, se ovan. En copied_pointer ska kunna initieras på tre sätt:

```
copied_pointer p1;                      // tompekare

copied_pointer p2(new Integer(1));      // pekare till Integer

copied_pointer p3(p2);                  // kopiering
```

Kopieringstilldelning för copied_pointer ska finnas:

```
p1 = p2;
```

I exemplet ovan ska objektet som p2 pekar på kopieras och pekaren till kopian ska lagras i p1. Ett eventuellt tidigare objekt som refereras av p1 ska städas bort.

När ett copied_pointer-objekt upphör att existera ska minnet för objektet, om det finns ett, återlämnas.

copied_pointer ska också ha move-konstruktor och move-tilldelningsoperator.

Operatorerna **operator*** (”dereferencing”) och **operator->** (”indirection”) ska kunna användas på copied_pointer-objekt med samma effekt som om copied_pointer-objekten vore vanliga pekare:

```
cout << p1->get_value() << endl;

p1->set_value(4711);
(*p1).set_value(17);

cout << *p1 << endl;    // om operator<< överlagrats för Integer!
```

Gör om copied_pointer till en klassmall (**template**), så att man kan deklarera smarta pekare för ”godtyckliga” typer av objekt:

```
copied_pointer<Integer> sp1(new Integer(1));

copied_pointer<string>  sp2(new string);
```

Det finns ett svårlöst problem här! Om skapandet av ett copied_pointer-objekt misslyckas, vem ska/kan ta hand om det dynamiska objekt som anges som initialvärde?

Det ingår *inte* i uppgiften att implementera generella möjligheter för att testa om copied_pointer-objekt är tompekare eller att kunna jämföra två copied_pointer med avseende på likhet/olikhet. Det leder alldeles för långt. En enkelt åtgärd är överlagra **operator**! och låta den returnera sant om den råa pekaren är en tompekare. Det gör det åtminstone möjligt att skriva uttryck som följande:

```
if (!p1) …          // ”om p1 är en tompekare…”
```

Övningsuppgifter för lektion 2-3

Uppgiften tar upp arv, initiering och destruering i klasshierarkier, polymorfi och operatoröverlagring. Övningen är förberedande för främst laborationsuppgift Kalylatorn.

Antag att vi i ett program vill hantera namn och att ett namn kan vara antingen ett enkelnamn, som Anna, eller ett dubbelnamn, som Anna-Maria. Vi vill kunna hantera namn och dubbelnamn enhetligt i många situationer men även kunna särskilja ett enkelnamn från ett dubbelnamn.

1. Definiera en klass Name för att hantera enkelnamn och en subklass till name som ska heta Double_Name för att hantera dubbelnamn. I Name ska en datamedlem av typen std::string finnas för att lagra (enkel)namnet. I Double_Name ska en datamedlemmen av typen std::string finnas för att lagra den andra delen av ett dubbelnamn, t.ex. Maria, medan den första delen, t.ex. Anna, ska lagras i Name-delen av Double_Name-objektet. Namnobjekt ska kunna initieras med C-strängar eller std::string:

```
Name n1("Marie");                    // Marie
Name n2(n1);
string s1("Claude");                 // Claude
Name n3(s1);

Double_Name nn1("Jean", "Claude");   // Jean-Claude
Double_Name nn2(nn1);
string s2("Marie");
Double_Name nn3(s2, s1);             // Marie-Claude
```

Följande tilldelningar ska vara tillåtna:

```
n1 = n2;
n2 = "Claudette";
string s("Jeanne");
n3 = s;

nn1 = nn2;
```

Move-semantik bör finnas, om det är meningsfullt.

2. Det ska också vara möjligt att skriva ut namn på en utström med **operator**<<, t.ex.

```
cout << n1 << endl;       // Skriver ut, t.ex., Marie
cout << nn1 << endl;      // Skriver ut, t.ex., Jean-Claude
```

Detta ska åstadkommas genom att endast definiera en instans av **operator**<<, gemensam för Name och Double_Name.

Ytterligare funktionalitet kan införas vid behov. [Tips: swap-funktioner kan vara användsbart.]

Genom att ange tomma strängar i initieringar och tilldelningar, se nedan, kan otillåtna namn skapas, t.ex. Det bortser vi från.

```
Name nn("");
nn = "";
```

Deklarera Name och Double_Name på en gemensam inkluderingsfil, Name.h, och separata definitioner av medlemsfunktioner på en gemensam implementeringsfil, Name.cc. Skriv ett testprogram för att testa Name och Double_Name.

3. Lägg till en medlemsfunktion clone(), som kan anropas för att skapa en kopia av det objekt som funktionen anropas för.

```
Name* p1 = new Name("Jean");
Name* p2 = p1->clone();
p1 = new Double_Name(""Jean", "Pierre");
p2 = p1->clone();
```

Syftet med en sådan funktion är att man ska kunna skapa korrekta kopior av objekt i en klass-hierarki, även då man refererar objekten via polymorfa pekare och inte på förhand vet exakt vilken typ det gäller och därför inte kan hårdkoda vilken typ av objekt som ska skapas.

4. Definiera **operator**>> för Name och Double_Name.

Det kan vara knepigt att konstruera **operator**>> för Double_Name men idiomet "skapa temporär och byt” kan vara ett sätt att lösa problemet, utan att **friend**-deklarera eller på annat sätt skapa åtkomst till privata medlemmar.

Övningsuppgifter för lektion 4

Klass- och funktionsmallar (operatorfunktioner).

1. Nedanstående enkla **struct** för att lagra ett **int**-värde är given:

```
struct Wrapper
{
  int value_;
};
```

Gör följande:

a) Gör om Wrapper till en mall med en typparameter T (typen för datamedlemmen value_).

b) Gör datamedlemmen value_ **private** – lämpligt att byta **struct** mot **class** – och inför åtkomstfunktionerna get_value() och set_value() för att läsa av respektive ändra value_.

c) Lägg till en konstruktor för att initiera ett Wrapper-objekt med ett värde av typen T. Om inget argument ges till konstruktorn ska defaultinitiering av value_ göras med ”defaultvärdet” för typen T.

d) Deklarerar ett objekt wrap av typen Wrapper<**int**> och initiera med t.ex. 4711.

e) Skriv kod för att skriva ut wrap:s värde sedan läsa in ett nytt värde till wrap.

f) Definiera **operator**<< som mallfunktion för att kunna skriva ut värdet av ett objekt av typen Wrapper<T> på en ostream.

```
cout << wrap << '\n';
```

operator<< förutsätts vara definierad för typen T.

g) Definiera **operator**>> som mallfunktion för att kunna läsa in ett värde till ett objekt av typen Wrapper<T> från en istream.

```
cin >> wrap;
```

operator>> förutsätts vara definierade för typen T.

h) För att få till det på samma sätt som ovan för objekt av typen Wrapper<vector<**int**>> behöver **operator**<< och **operator**>> definieras för vector<T>. Använd algoritmen copy, ströminteratorer och back_inserter (vid inläsning) för att implementera operationerna.

x) Xtremt kul xtrauppgift – deklarera en *explicit*-deklarerad operatorfunktion för att typomvandla ett Wrapper-objekt till T, prova

```
Wrapper<vector<int>> w;
// w fylls på med värden

vector<int> v = w;
```

explicit-deklarera sedan typomvandlingsfunktionen (som den bör vara) och modifiera koden på den sista raden ovan så att den tillåts.