LINKÖPINGS TEKNISKA HÖGSKOLA
Institutionen för datavetenskap
Avdelningen för programvara och system
Tommy Olsson 2013-05-21

Övningsuppgift för lektion 3

Uppgiften tar speciellt upp initiering, kopiering, tilldelning och destruering av klassobjekt samt operatoröverlagring. Uppgiften är förberedande för laboration 2.

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

```
class Integer
{
  public:
    Integer() noexcept = default;
    explicit Integer(int value) : value_{value} {}

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

  private:
    int value_{0};
};
```

Vi vill säkerställa att minnet alltid återlämnas då en pekare som pekar på ett sådant objekt 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 ”smartpekare”. En smartpekare är ett klassobjekt som kapslar en vanlig, rå pekare och vars destruktor ser till att minnet för objektet som den råa pekaren pekar på återlämnas. Det är önskvärt att en smartpekare 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 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 smartpekare.

Det finns olika modeller för smartpekare. I standardbiblioteket finns till exempel unique_ptr, som gör ”destruktiv kopiering” och ”destruktiv tilldelning”. Det innebär att kopieringskonstruktorn och kopieringstilldelningsoperatorn 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}};
unique_ptr<Integer> ap2{std::move(ap1)};
ap1 = td::move(ap2);
```

”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). Det kan bland annat vara intressant då man ska överföra pekare till funktioner och returnera pekare från funktioner.

Andra vanliga modeller för smartpekare är:

- ”copy on construct/assign” eller ”deep copy” – då en sådan smartpekare kopieras eller tilldelas görs en kopia av objekt som pekaren i källan pekar på (djup kopiering).
- ”copy on write” – då en sådan smartpekare kopieras eller tilldelas kopieras bara pekaren, dvs mottagaren och källans pekare pekar på samma objekt. Först om en operation utförs på en smartpekare 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 smartpekare 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 smartpekare, där syftet främst är att ta upp initiering, kopiering, tilldelning och destruering för klassobjekt samt operatoröverlagring.

Uppgiften

Konstruera en smartpekare kallad copied_ptr för att hantera objekt av typen Integer, se ovan. En copied_ptr ska kunna initieras på bland annat följande sätt:

```
copied_ptr p1;                          // defaultkonstruktor (tompekare)
copied_ptr p2{new Integer{1}};          // pekare till Integer
copied_ptr p3{p2};                      // kopieringskonstruktor
copied_ptr p4{nullptr};                 // uttrycklig initiering till tompekare
```

Dessutom ska *move-konstruktor* finnas. *Kopieringstilldelning* ska finnas:

```
p1 = p2;
```

I exemplet ovan ska objektet som p2 pekar på kopieras till p1. Ett eventuellt tidigare objekt som p1 ansvarat för ska städas bort. Även *move-tilldelningsoperator* ska finnas.

När ett copied_ptr-objekt upphör att existera ska minnet för objektet, om det finns ett, *destrueras*.

Operatorerna **operator*** och **operator**-> ska kunna användas på copied_ptr-objekt med samma effekt som om copied_ptr-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
```

Funktionerna *swap* och *get*, enligt vad som sagts tidigare, ska finnas.

En *typomvandlingsoperator* till **bool** ska finnas, ska kunna användas exempelvis:

```
if (p1) …           // om p1 inte är en tompekare…
```

Implicit typomvandling till och från copied_ptr ska *inte* vara tillåten.

Överkurs

Gör om copied_ptr till en klassmall (**template**), genom att byta Integer mot en malltypparameter T, så att man kan deklarera smarta pekare för godtyckliga typer av dynamiska objekt:

```
copied_ptr<string> sp{new string};
```