# Avancerad Programmering i C++

## Övning – Smartpekare I

I den här uppgiften ska du, givet en enkel klassdefinition för en smartpekartyp, först implementera de givna medlemsfunktionerna och sedan successivt utveckla klassen så att användningen, så långt möjligt, överensstämmer med vad som gäller för vanliga, råa pekare. Smartpekaren ska dock ha förbättrade egenskaper i några viktiga avseenden och implementera någon vanlig semantisk modell för smartpekare.

I grunden är det fråga om att designa en enskild icke-trivial klass och överlagra pekarrelaterade operatorer.

Det finns en motsvarande uppgift Smartpekare II som innebär design av en mer realistisk smartpekartyp i form av en klassmall, så att den lagrade typen kan varieras.

På sista sidan finns en lista över testfall. Observera att vissa testfall är relevanta endast om smartpekarklassen är en mall och därmed olika instanser för mer eller mindre kompatibla typer kan förekomma.

### Smartpekare

En smartpekartyp ska i grunden ha samma egenskaper som en vanlig pekartyp – initiering, kopiering, tilldelning, avreferering med `*` (*dereferencing*) och medlemsåtkomst med `->` (*indirection*). I vissa avseenden vill vi att smartpekare ska ha andra och dessutom bättre egenskaper, t.ex. att en smartpekare alltid initieras och att minnet för refererade objektet alltid återlämnas när smartpekaren upphör att existera. Andra viktiga frågor gäller kopiering och tilldelning, som kan implementeras enligt olika modeller. Andra aspekter är typning (t.ex. om och hur olika smartpekartyper ska kunna blandas i operationer) och att kunna använda konstanta och icke-konstanta smartpekare av en viss typ på samma sätt som vanliga pekare.

Det finns ett flertal implementeringsmodeller och optimeringstekniker för smartpekare. De mest kända är *djup kopiering (”copy-on-create/assign”)*, *destruktiv kopiering*, *referensräkning*, *referenslänkning* och *kopiering-vid-ändring* (”*copy on write*”). En specifik fråga när det gäller kopiering är om, och i så fall hur, kopiering kan göras för smartpekare till polymorfa objekt. En basklasspekare kan ju peka på ett objekt av subklasstyp och frågan är då om en korrekt kopiering kan göras.

Ytterligare ett exempel på en frågeställning är om det ska vara möjligt att initiera en smart pekare till tompekare och ska det i så fall finnas en defaultkonstruktor som gör det; ska det vara tillåtet att uttryckligen initiera med en tompekarkonstant, 0.

I standardbiblioteket för C++ ingår en smartpekartyp, auto_ptr (”deprecated” i C++0x och ersatt av andra smartpekaretyper). Denna är implementerad för *destruktiv kopiering*. När en auto_ptr kopieras till en annan auto_ptr kommer den förra att ”destrueras”, vilket innebär att den råa pekaren i till objektet överförs från källan till kopian och att pekaren i källan sätts till tomma pekaren. Exempel:

```
auto_ptr<int> sp1(new int(1));    // sp1 initieras med ett int-objekt.
auto_ptr<int> sp2(sp1);           // sp2 övertar objektet, sp1 blir tompekare.
sp1 = sp2;                        // sp1 övertar objektet, sp2 blir tompekare.
```

Motsvarande övertagande av pekaren sker t.ex. när en funktion som har en värdeparameter av typ auto_ptr anropas. I den här övningen ska du implementera annan semantik för smartpekare.

## Allmänt om uppgiften

Utgångspunkten är klassen smart_pointer nedan.

```
class smart_pointer
{
  public:
    explicit smart_pointer(int* ptr = 0);
    smart_pointer(const smart_pointer& rhs);
    ~smart_pointer();
    smart_pointer& operator=(const smart_pointer& rhs);
    int& operator*();
    int* operator->();
  private:
    int* ptr_;
    void copy(const smart_pointer& ptr);
};
```

Den typ av objekt som smartpekaren kan peka på är alltså **int** (som borde vara av en malltypparameter T). Medlemsfunktioner för initiering, kopiering, kopieringstilldelning, destruering samt de två grundläggande pekaroperationerna * (*dereferencing*) och -> (*indirection*) deklareras. Privat deklareras den råa pekare som ska peka på det objekt smartpekaren ska referera till (peka på) samt en hjälpfunktion copy(). Funktion copy() ska implementera semantiken för kopiering i samband med initiering, kopiering och tilldelning. Andra medlemsfunktioner som utför operationer som rör ägarskapet till det refererade objekt ska använda copy(), för att dela kod och göra det enkelt att byta kopieringssemantik.

Nedan ges exempel på deklarationer och uttryck som ska, eventuellt ska, eller inte ska vara tillåtna. För t.ex. jämförelser finns ett antal symmetriska fall men endast några exempel ges här.

```
smart_pointer sp1(new int(1));    // Initiering med rå pekare
smart_pointer sp2(sp1);           // Kopieringskonstruktor

sp1 = sp2;                        // Kopieringstilldelning

int* p = new int(2):
smart_pointer sp(p);

delete p;                         // Problem – destruktorn för sp ska göra detta också
delete sp;                        // Ska inte vara tillåtet

if (sp) …                         // Direkt tompekartest
if (!sp) …                        // Negerat direkt tompekartest.
if (sp == 0) …                    // Uttrycklig test om sp är en tompekare
if (0 == sp) …                    // Symmetrisk fall
if (sp != 0) …                    // Uttrycklig test om sp ej är tompekare
if (0 != sp) …                    // Symmetriskt fall

if (sp1 == sp2) …                 // Uttrycklig jämförelse om två smartpekare är lika
if (sp1 != sp2) …                 // Uttrycklig jämförelse om två smartpekare är olika
if (sp1 < sp2) …                  // Tänkbart att test map ordning (<, <=, >, >=) ska kunna göras

smart_pointer sp;
int* p = new int(3);
if (sp == p) …                    // Jämförelser med vanlig pekare map likhet
if (sp != p) …                    // Jämförelser med vanlig pekare map olikhet
```

Om en smartpekare pekar på ett objekt av klasstyp ska punkt- och piloperatorn kunna användas för att komma åt medlemmar.

```
spc->m                            // operator-> (m antas vara medlem av klasstypen i fråga)
(*spc).m                          // operator* (smartpekaren avrefereras först)
```

## 1. Implementering av de givna medlemsfunktionerna

Implementera medlemsfunktionerna i den givna klassen smart_pointer för ”copy on create/assign”, dvs djup kopiering.

Det är lämpligt att börja med copy, som ska implementera kopieringssematiken. Använd sedan copy i implementeringen av de andra medlemsfunktioner där kopiering ska göras.

- copy ska göra en kopia av objektet som parametern ptr refererar till och tilldela pekaren ptr_ i det aktuella objektet den pekare som copy returnerar. Funktionen ska inget returnera. Parametern ptr kan vara en tom pekare.
- Konstruktorn som tar en pekare av typen **int*** ska initiera smartpekarobjekt et med pekaren. Default-argument ska vara 0.
- Kopieringskonstruktorn ska implementeras för ”copy on create/assign”.
- Destruktorn ska återlämna minnet för det refererade objektet, om det finns ett sådant (ptr_ kan vara en tompekare, 0).
- Kopieringstilldelningsoperatorn ska utföra tilldelning enligt semantiken för ”copy on create/assign”.
- **operator***() ska returnera en referens till det refererade objektet.
- **operator->**() ska returnera pekaren ptr_.

**Testa** att de implementerade medlemsfunktionerna i grunden fungerar som de ska och hur mycket av önskad semantik som erhålls genom denna första implementering samt om det eventuellt finns oönskade effekter.

En fråga när undantag har behandlats: Vilka av medlemsfunktionerna i smart_pointer skulle kunna deklareras att *inte* kasta undantag, dvs med undantagsspecifikationen throws()? Eller kommentaren // throws().

## 2. Implementering av grundläggande tompekartest och likhetstest

**Ett enkelt sätt** att möjliggöra vissa tompekartest och likhetstest (== och !=) vore att definiera implicit typomvandling från smart_pointer till någon annan typ. Följande visar typomvandlingsoperatorer som mer eller mindre naturligt skulle kunna övervägas:

- **operator int***()
- **operator void***()
- **operator const void***()
- **operator bool**()

Implicit typomvandling till sådana typer skapar dock i många fall betydligt mer problem än det löser. Typomvandling kan dyka upp där det inte alls är önskvärt och till exempel göra det möjlig att applicera **delete** på en smart pekare, tillåta att olika typer av smartpekare blandas i uttryck eller att en smart pekare kan användas som om den vore en aritmetisk typ. Sådant skulle dels bryta mot semantiken för smartpekare och dels göra det möjligt att skriva ren nonsenskod.

Om man definierar både **operator int***() för att erhålla implicit typomvandling till den råa pekartypen, kan man även definiera **operator void***() för att eliminera möjligheten att använda **delete** på smartpekare. **delete**-uttryck skulle genom det bli tvetydiga, eftersom ingen av dessa två omvandlingar är ett bättre val än den andra. Detta löser dock bara vissa problem och framstår dessutom som konstlat.

Om man överlagrar **operator**! och låter den returnera **true** om smartpekaren är en tompekare, annars **false**, möjliggörs vissa tompekartest, t.ex.:

```
if (!sp) …
if (!!sp) …        // Ett krystat sätt att skriva if (sp) …
```

Ett test som inte går att få till med enkla medel (utan att använda implicit typomvandling) är *direkt tom-pekartest*:

```
if (sp) …
```

Det finns ett avancerat trick för att lösa detta utan tillåta **delete**, men det väntar vi med.

En bra lösning är att överlagra operatorerna !, == och !=. Det innebär en del kodning men det får anses rimligt eftersom vi erhåller mycket av det som önskas, utan några oönskade bieffekter.

**Implementera** alltså följande:

- **operator**!, så att den returnera **true** om en smart pekare är en tompekare, annars **false**.
- **operator**== och **operator**!=. Jämförelse med avseende på likhet resp. olikhet ska kunna göras mellan två smartpekare av samma typ, eller mellan en smart pekare och en motsvarande rå pekare av motsvarande typ, eller mellan en smart pekare och tompekarkonstant.

**Testa** operatorerna.

## 3. Smartpekare och const

För råa pekare har vi följande möjligheter att använda **const**-kvalificering:

```
int*             p = 0;                    // pekare till int
const int*       pc = 0;                   // pekare till konstant int
int* const       cp  = new int(1);         // konstant pekare till int
const int* const cpc = new const int(2);   // konstant pekare till konstant int
```

Reglerna för råa pekare då det gäller **const** demonstreras av programmet på filen pointer-const-tests.cc i http://www.ida.liu.se/~tao/Attentec/src/Smartpekare/

Vår nuvarande smart_pekare är typmässigt **int***. Vilka av ovanstående pekartyper kan smartpekaren alltså referera till? Testa!

Möjligheten att deklarera konstanta smartpekare, till exempel

```
const smart_pointer csp0;

const smart_pointer csp1(new int(1));
```

har hittills inte nämnts. Om du inte redan har tänkt på det, modifiera smart_pointer, så att man kan operera även på konstanta smartpekare med operationer som bör vara tillåtna på konstanta objekt.

För att designa en mer realistisk smartpekartyp, se Smartpekare II.

# Testning

### Grundläggande funktionalitet

- Initiering: konstruktor som tar rå pekare till objekt, även 0 (default)
- Kopiering: kopieringskonstruktor och kopieringstilldelningsoperator
- Destruktor: återlämna minnet för det refererade objektet
- Avrefereringsoperatorn, **operator***
- Piloperatorn, **operator->** (kompileringsfel om det refererade objektet ej är av klasstyp)

### Destruering

- Det ska inte vara möjligt att utföra **delete** på en smartpekare (kompileringsfel)

### Tompekartest

- Direkt tompekartest, t.ex. **if** (sp)
- Negerat direkt tompekartest, t.ex. **if** (!sp)
- Uttryckligt likhetstest för tompekare, sp == 0, 0 == sp
- Uttryckligt olikhetstest för tompekare, sp != 0, 0 != sp

### Likhets- och olikhetstest (== och !=)

- Jämföra två smartpekare av samma typ
- Jämföra en smartpekare och en rå pekare av motsvarande typ
- Jämföra två polymorfa smartpekare av olika subtyper
- Jämföra en polymorf smartpekare och en rå pekare av olika subtyper

### Bevara konstanthet

- Tilldelning i olika varianter av smartpekare, konstant smartpekare, konstant rå pekare, rå pekare till konstant och konstant rå pekare till konstant.

### Stark typning

Dessa operationer ska *inte* vara tillåtna (kompileringsfel):

- Jämföra två smartpekare av inkompatibla typer (de refererade objekten är av inkompatibla typer)
- Jämföra smartpekare med rå pekare av inkompatibel typ
- Tilldelning med två smartpekare av inkompatibla typer
- Tilldela smartpekare med rå pekare av inkompatibel typ
- Initiera smartpekare med rå pekare av inkompatibel typ
- Initiera smartpekare med annan smartpekare av inkompatibel typ

### Funktionsanrop

- Parameteröverföring: värdeöverföring, referensöverföring
- Returvärde: värderetur, referensretur