LINKÖPINGS TEKNISKA HÖGSKOLA
Institutionen för datavetenskap
Avdelningen för Programvara och system
Tommy Olsson

2013-08-12

Skapa, kopiera och destruera klassobjekt

Detta dokument tar upp viktiga, grundläggande saker att tänka på då man konstruerar klasser.

Initiera datamedlemmar i samma ordning som de deklareras

Datamedlemmar initieras alltid i den ordning som de deklareras i klassdefinitionen. Datamedlemmar av grundläggande typ eller pekare har dock ingen defaultinitiering. Skriv alltid initierare i konstruktorer i samma ordning som medlemmarna deklareras, för att överensstämma med den ordningsom de kommer att initieras. Man undviker då irriterande varningar från kompilatorn. Exempel (felaktigt!):

```
class Employee {
public:
   Employee(const string& firstname, const string& lastname)
     : firstname_{name}, lastname_{lastname},
       email_{firstname_ + ´.´ + lastname_ + ”@liu.se”} {}
private:
   string email_;
   string firstname_;
   string lastname_;
};
```

Eftersom email_ är deklarerad före firstname_ och lastname_ kommer email_ att initieras först och då använda de ännu ej initierade medlemmarna firstname_ och lastname_. Om definitionen för konstruktorn skrivs separat blir det ännu svårare att observera felet. Exekveringsfel kan bli följden.

I C++11 kan man initiera datamedlemmar redan i deras deklaration, vilket man bör utnyttja. Om det även finns en initierare i en konstruktor är det inget problem, i så fall är det konstruktorns initiering som gäller. Detta är speciellt intressant för klasser med flera konstruktorer och olika sätt att initiera objekt.

Anm. Anledningen till att designbeslutet att medlemmar initieras i deklarationsordning är att säkerställa en unik destrueringsordning. I annat fall skulle destruktorn behöva kunna destruera medlemmarna i olika ordning, beroende på vilken konstruktor som skapat ett objekt. Den extra bokföring som det skulle kräva vore inte acceptabel. Det bästa är om man kan undvika att en medlems initiering är beroende av andra medlemmars initiering. I exemplet ovan är problemet enkelt att rätta till, antingen deklareras email_ efter de andra eller så kan man använd parametrarna för att initiera email_ och , i det senare fallet, skriva initierarna i samma ordning som medlemmarna deklareras.

Vid härledning gäller samma regel, basklassubobjekt initieras i samma ordning som basklasserna deklareras och basklassubobjekt initieras före de ”vanliga” datamedlemmarna.

Använd hellre initiering än tilldelning i konstruktorer

Klassmedlemmar av klasstyp defaultinitieras. Om man inte initierar sådana medlemmar och använder tilldelning i stället, enligt följande

```
class Employee {
public:
   Employee(const string& firstname, const string& lastname)
   {
      firstname_ = firstname;
      lastname_ = lastname;
      email_ = firstname_ + ´.´ + lastname_ + ”@liu.se”;
   }
```

```
private:
   string email_;
   string firstname_;
   string lastname_;
};
```

kommer defaultinitiering ändå att göras motsvarande följande:

```
class Employee {
public:
   Employee(const string& firstname, const string& lastname)
     : email_{}, firstname_{}, lastname_{}
   {
      // Tilldelningarna enligt exemplet ovan
   }

private:
   // Deklarationerna enligt exemplet ovan
};
```

Medlemmarna initieras alltså av defaultkonstruktorn för string och tilldelas sedan med kopierings-tilldelningsoperatorn för string. Vanligtvis behöver en tilldelningsoperator göra mer än en konstruktor, eftersom den opererar på ett redan existerande objekt. Förutom att initerare ofta ger bättre läsbarhet finns det alltså även en effektivitetsaspekt för medlemmar med defaultinitiering.

Medlemmar som är konstanter eller referenser måste initieras. Det är inte tillåtet att använda tilldelning för sådana medlemmar, i analogi med att man i ett block inte kan deklarera en konstant, eller en referens, utan att initiera den och senare tilldela den ett värde.

Då en konstruktor ska allokera resurser, till exempel dynamiskt minne (**new**), är det vanligtvis bättre eller nödvändigt att göra det i kroppen för konstruktorn. Om en resursallokering misslyckas kan till exempel andra, redan erhållna resutser, behöva återlämnas och det kan visa sig omöjligt att hantera om man allokerar resurser i samband med att initierare utförs.

Basklassdestruktorer bör vara public och virtual eller protected och ej virtual

När man härleder klasser är ofta syftet att använda objekten polymorft och definitivt så om det finns vanliga medlemsfunktioner som deklareras **virtual**. Det finns två huvudalternativ då det gäller hur en destruktor i en klass som är basklass bör deklareras. Om objekt ska kunna deallokeras via en basklass-pekare måste basklassens destruktor vara **public**, för att det ska vara tillåtet (pekarens typ, den statiska typen, avgör), och **virtual** för att korrekt destruktor ska anropas initialt, dvs destruktorn för den dynamiska typen (objektets typ).

```
class Base {
public:
   virtual ~Base();
};

class Derived : public Base {
public:
   ~Derived(); // Implicit virtual, liksom även den kompilatorgenererade hade varit
};

Base* bp = new Derived;
…
delete bp;   // OK, ~Base() är public; virtual medför att ~Derived() anropas
```

I annat fall kan basklassens destruktor vara **protected** och inte **virtual**.

```
class Base {
public:
   // Bör inte finnas några publika konstruktorer – fristående Base-objekt kan inte destrueras
protected:
   ~Base();
};

class Derived : public Base {
public:
   ~Derived();
};

Base b;      // Fel, ~Base() är inte public, objekt kan inte skapas
Derived d;   // OK, ~Derived() är public (anropar ~Base som är protected)
```

Om en klass är både basklass och ska kunna användas som en konkret klass för att skapa fristående objekt måste dock destruktorn vara **public**.

En kompilatorgenererad destruktor är **public** och inte **virtual**, vilket inte passar på något av de två alternativen ovan. Det innebär att för härledda klasser måste vanligtvis en destruktor deklareras.

Destruktorer ska aldrig misslyckas

Destruktorer, funktioner som frigör resurser (till exempel **delete**) och funktioner som byter innehåll på två objekt (swap-funktioner i många fall), ska aldrig behöva misslyckas. En destruktor ska aldrig kasta undantag, **noexcept**.

Glöm inte kopiering

Det är lätt att glömma bort kopiering eftersom kompilatorn genererar kopieringskonstruktorn och kopieringstilldelningsoperatorn, om man inte deklarerar dem själv. Det finns i princip tre alternativ:

- Använd de kompilatorgenererade versionerna om de fungerar.
- Skriv både kopieringskonstruktorn och kopieringstilldelningsoperatorn om klassen ska ha kopiering men de kompilatorgenererade versionerna inte fungerar.
- Eliminera båda genom att deklarera dem **deleted**. Flyttversioner (se nedan) genereras inte.

```
class T {
private:
   T(const T&) = delete;
   T& operator=(const T&) = delete;
   …
};
```

Att eliminera kopiering medför en del begränsningar, till exempel att sådana objekt inte kan lagras i en standardcontainer men det skulle man antagligen i vilket fall inte vilja.

Flyttsemantik (move-semantik)

Normalt bör en klass, om den har kopieringskonstruktor och kopieringstilldelningsoperator även, ha *flyttkonstruktor (move-konstruktor)* och *flyttilldelningsoperator* (move-tilldelningsoperator).

```
T(T&&);               // Flyttkonstruktor för T
T& operator=(T&&);    // Flyttilldelningsoperator för T
```

En flyttkonstruktor flyttar innehållet från källobjektet till destinationsobjektet och ”nollställer” sedan källobjektet. En flyttilldelningsoperator flyttar innehållet från högeroperanden till vänsteroperanden och ”nollställer” sedan normalt källobjektet.

Kopiera och destruera konsekvent

Om man definierar någon av kopieringskonstruktorn, kopieringstilldelningsoperatorn eller destruktorn för en klass behöver man antagligen definiera även de övriga, eftersom dessa tre är relaterade inbördes:

- om man har en egen kopieringskonstruktor eller kopieringstilldelningsoperator ska man antagligen ha även den andra, eftersom båda bör ha likartade egenskaper
- om man eliminerar kopieringskonstruktorn eller kopieringstilldelningsoperatorn ska man antagligen eliminera båda, eftersom kopiering med någon av dessa antagligen inte ska tillåtas
- om man har egna kopieringsfunktioner behöver man antagligen ha en egen destruktor
- om man har en egen destruktor behöver man antagligen ha egen kopiering eller eliminera kopiering
- tillåts kopiering bör antagligen även möjligheten att flytta objekt finnas

Om kompilatorgenererade versioner fungerar är dessa att föredra framför egendefinierade, eftersom de kompilatorgenererade är ”triviala”, vilket kan ha stor betydelse vid exempelvis optimering.

Se upp med ”slicing”

”Slicing” av objekt uppstår när ett objekt av subklasstyp omvandlas till ett objekt av basklasstyp. Detta är en är lömsk omvandling eftersom den är osynlig och automatisk. Ett scenario kan se ut enligt följande.

```
class Base {
public:
   Base(const Base&);
   virtual ~Base();
   …
};

class Derived : public Base { … };

void fun(Base b);  // Kopieringskonstruktorn för Base gör kopian vid värdeöverföringen

Derived d;
fun(d);  // Funktionen fun anropas med d, ett objekt av subtyp till Base
```

Programmerarens avsikt är att funktionen fun() ska operera på objekt av typen Base och subtyper polymorft men glömde & efter Base i deklarationen av parametern b. När fun anropas med ett objekt av typen Derived, kommer kopieringskonstruktorn för Base att initiera b med en ”slajsad” kopia av d. Avsikten var att en referensöverföring skulle ske, vilket inte påverkar det refererade objektet.

Om man vill tillåta slicing under kontrollerade former kan man deklarera kopieringskonstruktorn **explicit**. Anropet av funktionen fun() ovan ger då ett kompileringsfel, eftersom användningen av kopieringskonstruktorn inte är explicit vid parameteröverföringen. Genom att deklarera kopieringskonstruktorn **explicit** elimineras dock all värdeöverföring, vilket bör vara förbehållet specialfall.

```
class Base {
public:
   explicit Base(const Base&);
   …
};

void fun(const Base& rb)
{
   Base b(rb);  // explicit användning av kopieringskonstruktorn för Base
   …
}.
```

Kopiering av polymorfa objekt

Ett sätt att kopiera polymorfa objekt (man känner inte statiskt till typen för objekt som ska kopieras) är att använda en virtuell kopieringsfunktion, som clone() nedan.

```
class Base{
public:
   …
   virtual Base* clone() const = 0;

protected:
   Base(const Base&);
   …
};

class Derived : public Base {
public:
   …
   virtual Derived* clone() const override { return Derived(*this); }

protected:
   Derived(const Derived& other) : Base{other} … { … }
   …
};
```

Det gäller att samtliga konkreta subklasser överskuggar clone() med en egen version som gör en korrekt kopia av subklassen ifråga. Märk alltid sådana överskuggningar med override, så uppmanas kompilatorn att kontrollera att det är korrekt.

Alla klasser bör ha en swap-funktion om det är meningsfullt

En swap-funktion byter innehåll på två objekt. Den är, förutom att byta innehåll på två objekt, även användbar för att implementera andra operationer. Deklarera en medlem och en icke-medlem, som då används i stället för std::swap.

```
class T{
public:
   …
   void swap(T& other);   // ska helst inte kasta undantag
   …
};

swap(T& a, T& b);
```

Ett exempel på användning av swap är för att implementera tilldelningsoperatorer:

```
T& operator=(const T& rhs)
{
   T temp{rhs);      // skapa en kopia – händer det dåliga saker händer det här – …
   swap(temp);       // … och byt med *this
   return *this;     // temp:s destruktor tar hand om skräpet
}
```

Se även nästa stycke för en variation på detta tema.

Egendefinierad kopieringstilldelningsoperator

När man ska skriva en egen kopieringstilldelningsoperator för en typ T bör den normalt deklareras på den ”klassiska formen”

```
T& operator=(const T& rhs);
```

Deklarera inte tilldelningsoperatorer **virtual**, det kan göra att tilldelning mellan olika typer av objekt möjliggörs. Det är i så fall bättre att ha en vanligt funktion, till exempel

```
virtual void assign(const T&);
```

operator= ska inte returnera **const** T&. Det förhindrar förvisso att skriva kod som (x = y) = z, men även att objekt av typen T kan lagras i standardcontainrar, vilka kräver att tilldelningsoperatorn returnerar T&.

Gör alltid kopieringstilldelningsoperatorn felsäker och gärna enligt den ”starka garantin”, dvs att inget minne läcker och inga objekt hamnar i ett odefinierat tillstånd.

Se till att alla tilldelningsoperator är ”självtilldelningssäkra”. Det bästa sättet att göra det är att använda idiomet ”skapa en temporär/kopia och byt”, se ovan, det gör operatorerna både starkt felsäkra och självtilldelningssäkra.

Anropa basklassers tilldelningsoperatorer uttryckligen och tilldela alla datamedlemmar.

Returnera alltid ***this**.

Undvik att anropa virtuella funktioner i konstruktorer och destruktorer

Virtuella funktioner beter sig inte virtuellt i konstruktorer och destruktorer. Om en basklasskonstruktor eller -destruktor skulle vilja använda en härledd klass får man tillgripa andra tekniker, eftersom sammansatta objekt konstrueras subobjekt för subobjekt, uppifrån och ner enligt klasshierarkin.

```
class Base{
public:
  Base() { memfun(); }
  virtual void memfun();
};

class Derived : public Base {
public:
  Derived() { memfun(); }
  virtual void memfun();
};

Derived d;
```

När objektet d ovan konstrueras utförs först konstruktorn för Base. Medan det sker är den dynamiska typen för objektet Base, inte Derived (**this**-pekaren har typen Base* **const**, inte Derived* **const**). Ett anrop av den virtuella funktionen memfun() i konstruktorn för Base kommer alltså att anropa Base::memfun(). Efter att konstruktorn för Base är avslutad utförs konstruktorn för Derived och i dess kropp har **this**-pekaren nu typen Derived* **const**. Att det är Base::memfun() som anropas i konstruktorn för Base är bra, eftersom medlemmarna som tillhör Derived ännu inte är initierade men inte det som skulle behövas. Medan konstruktorn för Base utförs finns inget sätt att avgöra om det är ett fristående objekt som håller på att konstrueras eller om det är ett subobjekt i ett större sammansatt objekt.

I vissa fall behövs så kallad ”post-construction”, vilket innebär att en virtuell funktion anropas direkt efter att det fullständiga objektet har konstruerats.