Kommentarer till boken

Boken kan betecknas som en blandning av programmering i C, C++98 och C++11. Innehållet sägs överensstämma med den nya standarden men det finns uppenbara luckor, fel och sådant som är *deprecated* (”har tagits avstånd från”). Sådant som är *deprecated* ska undvikas, det är antingen ersatt med något nytt, bättre, eller så finns det enbart kvar av bakåtkompatibilitetsskäl.

Uppdateringen med avseende på standardbiblioteket är klart bristfällig. I *Appendix C Standardalgoritmer* saknas de cirka 20 nya algoritmer som tillkommit i C++11.

Faktarutorna ”ska kunna tjäna som snabbreferens när man vill slå upp en viss språkkonstruktion”, sägs det. Så borde det vara men många faktarutor är både ofullständiga och en del innehåller fel.

En del kodexempel och rekommendationer strider mot standard och gängse konventioner, motiveringar och förklaringar är felaktiga. Detta gäller bland annat i avsnittet om operatoröverlagring.

Kodexemplen innehåller många felaktiga semikolon efter funktionskroppar. Koden går att kompilera om man inte kompilerar med strikt syntaxkontroll, men sådana semikolon är tomma deklarationssatser som inte ska finnas. Det finns även kod med så allvarliga fel att de kan få program att krascha under körning.

En hel del kodexempel baseras på C-fält och C-strängar och pekarexercis, vilket känns mer som C- än C++-programmering. Existensen av std::array, std::vector och std::string gör att behovet av C-fält och C-strängar numera är litet. Man behöver som C++-programmerare dock känna till C-fält och C-strängar.

Referenser tas upp på ett sätt gör dem svårbegripliga och författaren tycks inte riktigt ha begreppen klara för sig.

Datatypen **struct** har förpassats till kapitel 15 men är i vissa fall det självklara valet i stället för **class**, och borde ha använts i många fall redan från och med kapitel 7.

Svenska namn på variabler, funktioner, etc., ger ett amatörmässigt intryck och riskerar att göra koden oläsbar, då å, ä och ö ersätts a, a och o. Risken är påtaglig att namn får en helt annan innebörd än den avsedda, till exempel ”stall” (där man har hästar) i stället för ”ställ” (klockan), avlas (kan vara en aktivitet i stallet) i stället för ”avläs” (klockan). Alla reserverade ord, standardbiblioteksnamn, etc, är engelska ord, och då passar inte heller av den anledningen svenska (ibland obegripliga) namn in.

Detta är ett tidspressat försök att sammanställa olika slags synpunkter på boken ifråga, det lär finnas brister av olika slag…

2. Grunderna

Syntax för olika satser framställs felaktigt i faktarutorna. Ersätter man syntaktiska element med konkreta motsvarigheter kan det leda till syntaxfel. Till exempel ska inte *sats* följas av semikolon. Om semikolon ska finnas eller inte beror på den konkreta typen av *sats* och ingår i så fall i syntaxen för satsen ifråga.

Faktarutor kapitel 2

21, *Variabeldeklarationer*: Man bör inte initiera flera variabler eller konstanter i samma deklaration. Åtminstone en kommentar om det är inte är god programmeringsstil hade varit på sin plats.
Begreppet ”konstant variabel” får väl anses vara motsägelsefullt och förekommer inte i C++. (Begreppet finns men används till exempel när man i experiment har värden som kan varieras men hålls konstanta under det att experimentet utförs).

25, *Några vanliga manipulatorer i utskrifter*: Det kunde påpekas att <iomanip> endast krävs då manipulatorer som tar argument, som setw(*n*), används.

28, *Standardfunktioner i biblioteket cmath* och *Aritmetiska funktioner i cstdlib*: Det finns betydlig fler funktioner än de som anges.

31, ***if**-satsen, olika former*: När man beskriver syntax formellt bör man vara noggrann med detaljerna. Idén är ju att man ska kunna ersätta *uttryck* och *sats* med konkreta motsvarigheter och då kan inte *sats* följas av ett semikolon. Om vi byter ut sats mot något mycket vanligt i detta sammanhang, en block, **if** (e) {…}; **else** {…}; blir det syntaxfel (”else utan föregående if”).
Det finns endast två former av **if**-sats, med eller utan **else**, övriga ”former” som visas i faktarutan är bara variationer, varav ett par kan anses vara exempel på mindre bra kodningsstil (klamrar endast i den ena av grenarna i en **if-else**-sats). Det hade varit mer intressant att visa hur man bör utforma nästlade **if**-satser.
I en del satser ingår semikolon i syntaxen, till exempel i uttryckssatsen (*uttryck*;), **break**-, **continue**-, **return**-, **goto**-, **do**-satsen samt en del deklarationssatser.

32, *Jämförelseoperatorer*: Specifikt *likhetsoperatorer* (==, !=) och *relationsoperatorer* (<, <=, >, >=).

36, ***while**-sats, olika former*: Det ska det inte finnas något semikolon efter *sats* och egentligen finns det bara en form av **while**-sats, den till vänster. Den till höger får man om man ersätter *sats*, utan semilon, med ett block, {…}.

44, ***for**-satsen, fullständig version*: Första delen, *init*, är egentligen en sats (där semikolon ingår). Det kan antingen vara en *deklarationssats* där variabler deklareras, eller en *uttryckssats* (även tom sats då endast semikolonet ingår).

56-57, *Operationer för standardklasserna som beskriver sekvenser*: (en enklare och bättre rubrik vore *Operationer för sekvenscontainrar*) Saknas de nya emplace-operationerna i C++11 för att sätta in: emplace_front(*arg*) och emplace_back(*arg*) för deque och list, emplace_front(*arg*) för forward_list och emplace_back(*arg*) för vector. Det finns även emplace-funktioner som tar en iterator och *arg*. Argumentet *arg* kan vara inget, ett eller flera värden, vilka ska passa en konstruktor för den typ av element man har i containern. Med emplace-funktionerna skapas objekt direkt i containern, utan att först behöva skapa ett objekt i en separat variabel, initierad denna med *arg*, och sedan kopiera variabeln till containern.

Övrig text och kodexempel i kapitel 2

20, stycket mitt på sidan "*Om man inte initierar…*”: Huruvida en variabel initieras automatiskt eller ej beror på *var* den deklareras. Globala och statiska lokala variabler initieras alltid automatiskt, till exempel nollställs variabler av aritmetisk typ och objekt av klasstyp defaultinitieras.

20, rad 11 nerifrån: Det finns många liknande exempel i boken och de kan förhoppningsvis förklaras av att det sparar utrymme. Variabler som initieras bör ha sin egen deklaration, vilket är viktig för läsbarheten), Följande deklaration.

```
int dagantal=0, dagspris, tot_pris=0;
```

bör skrivas

```
int dagantal = 0;
int total_pris = 0;
int dagspris;
```

45, *2.7 Fält*: Sådana enkla C-fält behöver man känna till men deras användning är liten i C++11, eftersom containertyperna array och vector och strängtypen string med fördel kan användas i de allra flesta fall där C-fält används i C och i viss mån i C++98 (array kom först i C++11). Kommandoradsargunment (11.6) framtvingar användning av teckenfält, C-strängar. Boken bygger många exempel på C-fält eller C-strängar, där man kanske hellre sett annat.

52, *2.8 Sekvenser*: Rubriken *Sekvenscontainrar* vore mer passande. Det är inte nödvändigt att känna till hur C-fält fungerar för att förstå hur *vissa* av sekvenscontainrarna fungerar; det man behöver förstå kan man likaväl förklara direkt för containern ifråga.

53, de två kodexemplen längst ned: När man använder en *initierarlista* (*brace-initializer-list*), för att initiera exempelvis en vector, finns det generella motiv (tro på det:) för att föredra formen utan ”=”, dvs

```
vector<double> v3{ 1, 2, 3, 4, 5 };
```

Värdena kunde kanske anges med korrekt typ, **double**, dvs { 1.0, 2.0, 3.0, 4.0, 5.0 }.

54, rad 8: Om elementen i en array initieras eller inte beror på typen av element, om elementen är av klass-typ kommer de att initieras av sin defaultkonstruktor. I en defaultinitierad array<std::string, 10> kommer alltså de 10 string-objekten att initieras till tomma strängar.

58, *2.9 Den förenklade for-satsen*: I referensmanualen kallas denna *range-based for statement*, som på svenska skulle kunna kallas *intervallstyrd for-sats*, en kanske mer talande beteckning är *förenklad*. Denna **for**-sats förutsätter att elementen som satsen ska iterera över bestäms av iteratorer eller pekare. För en container *c* alltså *c*.begin() och *c*.end(), för ett C-fält *a* med dimension *N*, *a* respektive *a*+*N*, och för andra intervall *x* att funktionerna begin(*x*) och end(*x*) returnerar iteratorer/pekare som anger början och slut.

59, kodexemplet högst upp: Den liknelsen är felaktig och fungerar speciellt inte om e är en referens, eftersom referenser alltid måste initieras då de initieras och sedan inte kan bindas om. En mer korrekt liknelse är följande, om vi tar det andra kodexemplet med ”**for** (**int**& e : f)” på sidan 59:

```
for (int i = 0; i < 5; i++)
{
   int& e = f[i];  // semantiskt deklareras och initieras e på detta vis
   e = 0;
}
```

60, *2.10 Fel i program*: Ytterligare en vanlig typ av fel man kan drabbas av är *länkfel* (*länkningsfel*), vilket innebär att alla programdelar, typiskt funktioner, inte hittas då det körbara programmet ska sättas ihop. En orsak kan vara att ett program är uppdelat på flera filer och man glömt att ta med någon fil då programmet ska länkas. En annan orsak kan vara att deklarationen för en funktion (som använts vid kompilering av kod som anropar funktionen) inte stämmer överens med funktionsdefinitionen; antal parametrar, parametrarnas typer, **const** för medlemsfunktionerna kanske skiljer.

3. Tecken och texter

Ett plus är att string behandlas före C-strängar men mycket som rör C-strängar kunde kanske vänta till kapitel 15, liksom även en del av det som rör teckenkodning. Något som saknas i boken är hur man omvandlar mellan numeriska typer (**int**, **double**, etc.) och motsvarande strängrepresentation (string). I den nya standarden finns många nya funktioner för att göra sådana omvandlingar (tillhör <string>). Överlagrade funktioner omvandlar ett värde *x* av numerisk typ till string.

```
string to_string(x);
wstring to_wstring(x);
```

Följande funktioner omvandlar från sträng till angiven returtyp (det finns fler argument som kan anges, för olika syften; s kan vara string eller wstring).

```
int stoi(s);
long stol(s);
unsigned long stoul(s);
long long stoll(s);
unsigned long long stoull(s);
```

```
float stof(s);
double stod(s);
long double stold(s);
```

Omvandling av ett enskilt siffertecken (**char**) till motsvarande **int**-värde handkodas på ett plattformsoberoende sätt som c - ’0’ (c har typ **char**).

Övrig text och kodexempel i kapitel 3

84, klassen alpha: För att få till korrekt bokstavsordning för strängar där svenska bokstäber kan ingå på ett lite med standardmässigt sätt, kan man definiera en *traits-klass* för svenska bokstäver genom att härleda från std::char_traits och definiera om funktionerna lt() och compare().

```
struct sv_char_traits : public std::char_traits<char>
{
   static bool lt(const char_type& c1, const char_type& c2)
   {
      if (std::strchr("ÄÅäå", c1) && std::strchr("ÄÅäå", c2))
         return std::strchr("ÅÄåä", c1) < std::strchr("ÅÄåä", c2);
      return static_cast<unsigned char>(c1) < static_cast<unsigned char>(c2);
   }

   static int compare(const char_type* s1, const char_type* s2, size_t n)
   {
      for (std::size_t i = 0; i < n; ++i)
         if (lt(s1[i], s2[i]))
            return -1;
         else if (lt(s2[i], s1[i]))
            return 1;
      return 0;
   }
};
```

Skapa sedan en instans av std::basic_string för **char** och sv_char_traits och överlagra << och >> för att kunna skriva och läsa sv_string:

```
typedef std::basic_string<char, sv_char_traits<char>> sv_string;

std::ostream& operator<<(std::ostream& os, const sv_string& s);
std::istream& operator>>(std::istream& is, sv_string& s);
```

Detta löser dock inte problemen med hantering av svenska bokstäver i andra sammanhang.

89, *3.4 Teckenfält*: Behovet av C-strängar är i princip litet i C++, i och med att std:.string finns. En anledning till att behöva känna till dem är att kommandoradsargument är tillgängliga som C-strängar, main(**int** argc, **char*** argv[]). För övrigt får behovet anses vara begränsat till undantagsfall då det gäller grundläggande programmering.

4. Funktioner

Faktarutor har lagts in långt innan allt som de bör ta upp för att kunna vara någorlunda kompletta har behandlats. Detta gör dem ofullständiga och missvisande. I många av exemplen används C-fält, vilket gör koden i viss mening omodern som exempel på C++-kod.

Faktarutor kapitel 4

105, ***return****-sats*: I en funktion som inte ska returnera något (beskrivs först på sid 114 ff) utelämnas *uttryck*. Det finns även formen ”**return** *brace-init-list*;” för funktioner som returnerar värden på formen {x, y, z} (en *brace-init-list*, som inte är ett *uttryck*). Faktarutan borde komma efter 4.3.

107, *Funktionsanrop*: I punkt 3 bör vara ”Funktionen avslutas i en **return**-sats *eller då exekveringen når funktionens slut*.” Faktarutan borde komma efter 4.3.

126, *Uppdelning av program i flera filer*: Påståenden om filernas innehåll baserat på vad som hittills behandlats i boken, vilket blir missvisande. En inkluderingsfil kan typiskt innehålla en klassdefinition och deklarationer för tillhörande vanliga funktioner.

131, *Referensparametrar*: Parametrar av typen *referens till konstant*, **const** typ&, är så viktiga att de bör betraktas som en specifik parametertyp. Den typ av referenser som behandlas är enbart *lvalue-referenser*, *rvalue-referens* är också en möjligthet. ”Fält är redan en slags referenser.” kanske borde uttryckas annorlunda.

Övrig text och kodexempel i kapitel 4

117, sista stycket: Det verkar inte sägas någonstans att *block* är en sats, *sammansatt sats*, som utgör en sekvens av satser (en av de tre *strukturprimitiverna – sekvens*, *selektion* respektive *iteration*).

118, näst sista stycket: *deklarationsområde, declarative region*, är inte helt synonymt med *scope* (*räckvidd*). Med deklarationsområde avses det block, namnrymd eller klass i vilket ett namn är deklarerat. Den *potentiella* räckvidden för ett namn börjar typiskt där den deklaration som inför namnet är avslutad (*point of declaration*) och sträcker sig till slutet av det deklarationsområdet. Namn som deklareras utanför ett block, namnrymd eller klass är deklarerade i *global namnrymd*.

122, *4.5 Uppdelning av program*: C-strängar kunde gärna ha undvikits.

126, *4.6 Referensparametrar*: Det borde påpekas att det finns två slags referenser, *lvalue-referenser* och *rvalue-referenser*, och att enbart lvalue-referenser enbart behandlas i detta avsnitt men att rvalue-referenser är också en intressant och viktig parametertyp.

134, *4.7 Parametrar med defaultvärden*: *defaultargument* är den korrekta beteckningen.

138, *4.9 Alternativ syntax för funktioner*: Detta blir obegriplig när det inte sätts in i ett meningsfullt sammanhang, detta borde ha sparats till senare (kapitel 14. Templates till exempel). Typexemplet är en mallfunktion med olika parametertyper, som bestäms av mallparametrar, och där returtypen beror av parametertyperna och kan bestämmas av någon operation på parametrarna, där **decltype** kommer till användning. Exempel:

```
template<typename T1, typename T2>
auto fun(const T1& x, const T2& y) -> decltype(x + y)  // x+y beräknas inte i denna kontext
{
   return x + y;
}
```

Returtypen bestäms av typen för uttrycket a + y och det kan alltså vara T1 eller T1. Om T1 är **int** och T2 är **double** ska resultattypen vara T2, om tvärt om ska resultattypen vara T1.

5. Typer

Väl mycket fokus på pekare i kombination med C-fält och C-strängar samt pekararitmetik. En hel del sådant skulle kunna flyttas till kapitel 15. I modern C++-programmering bör man använda array, vector och string, om inga speciella skäl föreligger för att använda C-fält och C-strängar. Referenser är definitivt inte författarens starka sida…

Faktarutor kapitel 5

169, *Minnesallokering*: Det finns även en tredje form av **new**, kallad ”placement new”. Den anropas med adressen till ett redan befintligt minnesutrymme, **new** (*adress*) *typ*, och används för att konstruera ett objekt av typen *typ* i det minnesutrymmet. Om *typ* är en klasstyp körs en konstruktor som skapar ett objekt i minnesutrymmet. Placement **new** används typiskt för att separera allokering av minnes från konstruktion av objekt som ska lagras i minnet, till exempel för att undvika onödig defaultinitiering.

Övrig text och kodexempel i kapitel 5

150, första stycket och kodexemplet: Mer C++-mässig variant är <limits> och mallen numeric_limits.

```
numeric_limits<int>::min()// minsta värdet för int
numeric_limits<int>::max()// största värdet för int
```

150, **sizeof**: **sizeof**(**char**) är alltid 1, för övrigt implementationsberoende. Det finns även **sizeof**..., som har med *variadic templates* att göra, en ny och mycket intressant mallkonstruktion som inte nämns i boken.

152, första stycket och kodexemplet: Mer C++-mässig variant är <limits> och mallen numeric_limits.

```
numeric_limits<double>::digits10
numeric_limits<double>::max()
numeric_limits<double>::min()
```

152, *5.3 sizeof-operatorn*: Det finns även varianten **sizeof**... (används i samband med så kallade *variadic templates*, vilket inte tas upp i boken).

159, kodexemplet längst ner: I C++11 finns hjälpfunktionerna begin() och end() för så kallad *range access* (”intervallåtkomst”). Kodexemplet längs ned på sidan 159 kan med dessa och **auto** skrivas betydligt mer läsbart; inkludera <iterator> för att få tillgång till dessa hjälpfunktioner:

```
for (auto p = begin(f); p != end(f); p++)
   *p = 0;
```

Dessa hjälpfunktioner får man även med genom inkludering av <array>, <deque>, <forward_list>, <list>, <map>, <regex>, <set>, <string>, <unordered_map>, <unordered_set> och <vector>.

170, *5.4.6 Vanliga misstag*: Ett sätt att undvika en del problem associerade med pekare kan vara att använda ”smarta pekare” (*smart pointers*), vilka det finns tre slags i standardbiblioteket (unique_ptr, shared_ptr och weak_ptr).

174*, 5.5 Referenser*: ”temporärt uttryck” – vad är ett ”tillfälligt uttryck”? Temporära *objekt* är däremot ett viktigt begrepp i C++, med stark koppling till den nya typen av referenser, *rvalue-referenser* (&&).
Referenser är *inte* speciellt ”nära besläktade med pekare”. Allt skiljer egentligen, utom att i vissa situationer kan en referens innehålla adressen till det refererade objektet (kompilatorns val i den situationen). Initiering och användning av referenser skiljer helt från det som gäller för pekare.

175, *5.5.1 Referenser till variabler*: Detta avsnitt beskriver *lvalue-referenser*, dvs referenser som kan bindas till ett *lvalue*. Begreppet *lvalue*, tillsammans med *rvalue* (för att förenkla lite) används för att kategorisera uttryck. Ett *lvalue* (lvalue-uttryck) anger antingen en *funktion* eller ett *objekt*.
Exempel: namnet på en variabel, när det används i ett uttryck, är ett lvalue; om p är en pekare är uttrycket *p ett lvalue som refererar till det objekt eller den funktion som p pekar på; att anropa en funktion vars returtyp är en lvalue-referens (T&) är ett lvalue. Begreppet *lvalue* kommer ursprungligen

från ”något som kan stå till vänster (*left*) i en tilldelning”; något man kan ta adressen till det.
Namnregeln: Om något har ett *namn* är det ett *lvalue*, annars är det ett *rvalue*.
Strunta i alla jämförelser med pekare – en referens till ett objekt är helt enkelt ett annat namn för objektet (hur det går till behöver man inte bekymra sig om) och när man använder en referens är det refererade objektet man opererar på.

175, andra stycket nerifrån: Det finns *inga* undantag från regeln att referenser alltid måste initieras, det är enbart en fråga om variationer i syntax och semantik. För en vanlig referensvariabel ser vi uttryckligen en initierare i deklarationen, men det betyder inte att referenser som är funktionsparametrar, klassmedlemmar, etc., inte initieras, bara därför att vi inte ser en uttrycklig initierare i deklarationen. En referensparameter skapas då funktionen anropas och initieras av motsvarande argument. En klassmedlem av referenstyp skapas då klassobjektet skapas och initieras av en medlemsinitierare i den konstruktor som körs. En referens som deklareras i styrhuvudet på en ”förenklad” **for**-sats deklareras semantisk *inuti* satsdelen för **for**- satsen och det innebär att den (semantiskt) återskapas och initieras i varje iteration.

176, *5.5.2 Referenser till temporära uttryck*: Ska vara *temporära objekt* – en referens anger antingen ett objekt eller en funktion. Rubriken *Rvalue-referenser* hade nog varit att föredra i vilket fall, eftersom man inte kan sätta likhetstecken mellan rvalue-referenser och temporära uttryck, även om det finns en stark koppling. Ett *rvalue* (rvalue-uttryck) kan vara ett *prvalue* (pure rvalue) eller ett *xvalue* (”eXpiring value”). Ett *prvalue* kan vara ett temporärt objekt eller ett värde som inte är associerat med ett objekt. Att anropa en funktion vars returtyp inte är en referens är ett prvalue; värdet av en litteral som 4711, 3e5 eller **true** är ett prvalue. Ett *xvalue* refererar till ett objekt som *vanligtvis* är nära slutet av sin livstid; att anropa en funktion som returnerar en rvalue-referens är ett xvalue:

```
string s1 = "C++";
string s2;

s2 = std::move(s1);  // move returnerar string&& (vilket gör att move-tilldelning kan göras)
```

Exempel på referenser med vanliga variabler skapar mest ett förvirring. En viktig användning av rvalue-referens är som parametertyp (T&&), eftersom en sådan parameter matchar ett argument som är ett temporärt objekt bättre än någon annan parametertyp (speciellt T& eller **const** T&). Den förväntade kombinationen av överlagrade funktioner när rvalue-referens är inblandad är

```
void fun(T&&);        // väljs om argumentet är ett temporärt objekt – move-semantik används typiskt
void fun(const T&);   // väljs i annat fall – kopieringssemantik används typiskt
```

Dessa kan även överlagras med nedanstående (men *inte* med fun(T), för då uppstår tvetydighet)

```
void fun(T&);
```

och då skulle T& väljas om argumentet är ett lvalue (exempelvis en variabel av typ T), T&& väljas om argumentet är ett rvalue (till exempel ett temporärt objekt av typ T) och **const** T& (till exempel om argumentet är en konstant av typ T).

177, andra stycket innan 5.6: Det finns inga undantag, se kommentaren till motsvarande på sid 175.

177, *5.6 Komplicerade deklarationer – typedef*: Ett ofta mer lättläst alternativ är *alias-deklaration*:

```
using FP = void(*)(double);  // FP är pekare till funktion som tar en double och returnerar inget
```

182, näst sista stycket: Typomvandlingsoperatorerna static_cast, reinterpret_cast, dynamic_cast och const_cast kunde ha tagits upp först i 5.8, eftersom de vanligtvis är att föredra med tanke på att det finns en del kontroller av att de används i rätt sammanhang.

196, kodexemplet med tuple: Det finns ett smidigt sätt att ta emot flera returvärden från en funktion direkt i motsvarande variabler med hjälp av funktionen tie():

```
bool   reella_rotter;
double r1, r2;

tie(reella_rotter, r1, r2) = rotter(a, b);
```

7. Klasser

En så grundläggande sak som att medlemsfunktioner som inte ändrar på objekt ska **const**-deklareras tas upp först i senare kapitel. Det gör att kodexemplen i detta kapitel är felaktiga exempel på hur kod ska skrivas.

struct borde ha nämnts som ett ibland bättre val än **class** då en klasstyp ska konstrueras, även om exempel på sådana klasser inte dyker upp förrän i kapitel 12 (funktionsobjekt) och kapitel 13 (list- och trädnodtyper). Konventionen är att **struct** väljs om alla medlemmar ska vara synliga, dvs **public**, vilket är underförstått för **struct**. Ett sådant val mellan **class** och **struct** ger en direkt indikation i koden om förväntade egenskaper. Faktarutorna är lika giltiga för **struct** som för **class**, med några små skillnader då det gäller underförstådd tillgänglighet för medlemmarna.

Faktarutor kapitel 7

224, *Definition av medlemsfunktioner*: att **inline**-deklarera medlemsfunktioner får vissa konsekvenser relaterade till kompilering/länkning, som kanske borde nämnas?

229, *Uppdelning av klasser på flera filer*: Ett alternativ till att placera inline-funktioner i den ordinarie inkluderingsfilen är att placera dem i en speciell definitionsfil (till exempel med suffix .icc) och låter den ordinarie inkluderingsfilen (.h) inkluderar denna.

231, *Deklaration av konstruktorer*: Egentligen har konstruktorer inget namn, sättet att deklarera dem är speciell deklarationssyntax.

232, *Definition av konstruktorer*: Initierarna i en medlemsinitierarlista ska anges i samma ordning som datamedlemmarna är deklarerade i klassen. Det är deklarationsordningen som bestämmer initieringsordningen och man undviker, om inte annat, varningar från kompilatorn om man följer denna ordning.

237, *Kopieringskonstruktor*: Det finns även formen 'X x1{x2};'. Formen (olika parenteser eller =) är vanligtvis betydelselös men inte alltid då initieraren eller objektet som ska initieras är av klasstyp. Formen med = kallas *kopieringsinitiering* (copy initialization).

Kopieringsinitiering kan i vissa fall medföra att move-konstruktorn anropas i stället, till exempel vid retur från funktion. Formerna med () eller {} kallas *direktinitiering*. För klasser, något förenkla gäller:
1) om initieringen är direktinitiering beaktas enbart konstruktorer och den som passar bäst anropas. Om ingen konstruktor passar eller flera passar lika bra är initieringen "ill-formed".
2) om initieringen är kopieringsinitiering och typen för källan är samma klass som destinationen (eller en klass härledd från destinationstypen) tas endast konstruktorer under beaktande och den som passar bäst anropas. Om ingen konstruktor passar eller flera passar lika bra är initieringen "ill-formed".
3) för övrigt (dvs för de återstående kopieringsinitieringsfallen) undersöks om det finns en typomvandling som kan omvandla från källtypen till destinationstypen. De typomvandlingar som kan komma ifråga rangordnas och den som passar bäst väljs. Om ingen passande typomvandling finns eller flera passar lika bra är initieringen "ill-formed". Om den typomvandlingsfunktion som väljs är en konstruktur skapas ett temporärt objekt och detta används sedan för att direktinitiera enligt ovan. I vissa fall kan ett sådant temporärt objekt elimineras och initieringen sker då direkt i det objekt som ska initieras.

Sista kommentaren i rutan: Vad är "vanlig tilldelning"? Initiering och tilldelning är två helt skiljda saker.

249, *Destruktor*: Har egentligen inget namn, sättet att deklarera är speciell deklarationssyntax.

Övrig text och kodexempel i kapitel 7

218, klassen Klocka: Att deklarera medlemsfunktioner som är åtkomstfunktioner (ändrar inte på objektet) **const** är grundläggande och mycket viktig. Att ta upp detta först i kapitel 8 är olyckligt och alla klassexempel i kapitel 7 är därför i princip felaktigt konstruerade.

222, implementeringen av funktionen Klocka::skriv: onödigt att upprepa setfill('0') för varje värde som ska skrivas ut. Ändrar man utfyllnadstecknet kommer det att gälla till dess det ändras igen. Det tidigare utfyllnadstecknet bör också återställas:

```
char c = cout.fill('0');
   cout << setw(2) << t << ':' << setw(2) << m;
if (skriv_sek)
   cout << ':' << setw(2) s;
cout.fill(c);
```

223, *första stycket*: Sådana namn kallas *qualified names* (”kvalificerade namn”?)

223, tredje stycket och framåt: Kompilerings- och länkaspekter borde kanske nämnas redan här. Förklaras delvis på sid. 229.

229, *7.3 Konstruktorer*: Bakvänt att lägga fokus på vad som inte har varit tillåtet? Visa först vad som gäller i C++11, sedan kan man kommentera att detta inte var tillåtet i den gamla standarden.

230: Konstruktorer och destruktorer har inga namn, det vi ser är speciell deklarationssyntax.

232, kodexemplet nere på sidan: Att initiera variabler i en multipeldeklaration är inte god stil, varje objekt som har en initierare ska ha en egen deklaration.

235, exemplet högst upp på sidan och förklaring till vad som händer då defaultkonstruktorn utförs: Man skulle kunna visa vad som i princip sker med följande kodexempel:

```
Flight() : no(), dep(), arr() {}
```

238, *sist stycket*: Parametertypen väljs helt riktigt normalt som **const** X& men förutom X& kan den även vara **volatile** X& eller **const volatile** X&. I och med att rvalue-referenser (X&&) införts i C++11 har en tidigare vanlig anledning till att använda X& utgått.
Anm 1. Det får finnas fler parametrar men samtliga dessa måste i så fall ha defaultargument.
Anm 2. På samma sätt är en konstruktor en move-konstruktor om den har en (första) parameter av typ X&&, **const** X&&, **volatile** X&& eller **const volatile** X&&.
Anm 3. En deklaration av en konstruktor för en klass X är ”ill-formed” om dess första parameter är av typ X (med eventuell cv-bestämning).

239, det första kodexemplet: Notera att det enda man kan göra på ett dynamiskt fält med dimension 0 är **delete**[].

241, *7.3.5 Move-konstruktorer, första stycket, första raden*: Ersätt ”referenser till temporära uttryck” med ”rvalue-referenser”. Vanligtvis anger en rvalue-referens ett temporärt objekt men det är inte säkert. Däremot ska man kunna använda rvalue-referensen som om den anger ett temporärt objekt, speciellt att man ska kunna använda move-semantik.

243, *fjärde stycket (mitt på sidan), sista meningen*: Att ”Den är alltså en referens till ett temporärt objekt” kan man inte förutsätta.

243, *näst sista stycket, rad 2*: Stryk ”temporära”.

243, *näst sista stycket, rad 3*: Ändra ”pekaren p i det temporära objektet” till ”pekaren p i objektet v”.”

243, *näst sista stycket, rad 3*: Ändra andra förekomsten av ”det temporär objektet” till ”objektet v”.

243, *näst sista stycket, sista meningen*: Om objektet v säkert hade varit ett temporärt objekt hade detta varit en helt korrekt beskrivning. Det är dock riktigt att pekaren i objektet som resurserna flyttats från ska tilldelas nullptr. Objektet v ska vara destruerbart men det bör även vara fortsatt användbart (tilldelningsbart och kopieringsbart) i det fall det inte är ett temporärt objekt.

247, *tredje raden från slutet*: Egentligen inte viktigt i detta sammanhang, men en destruktor har inget namn, detta är speciell deklarationssyntax.

240, *implementeringarna av medlemsfunktionerna avlas och andra*: En kommenteras kunde vara att i dessa funktioner bör fel vid normal körning signalleras med att kasta undantag, range-error?

241, *7.3.5 Move-konstruktorer*: Den nya typen av referenser (&&) bör inte kallas ”referens till temporära uttryck”, utan *rvalue-referens*.

Begreppet ”temporära uttryck” är okänt, en tolkning skulle kunna vara ”uttryck som skapar temporära objekt” men det tillhör inte konventionell jargong. Den kategorisering som används för uttryck i C++ är *lvalue*, *xvalue* och *prvalue* samt *glvalue* (antingen lvalue eller xvalue) och *rvalue* (antingen xvalue eller prvalue). Ett lvalue anger antingen ett objekt eller en funktion. Ett xvalue anger ett objekt; typexemplet är anrop av en funktion som returnerar en rvalue-referens. Att anropa en funktion som *inte* returnerar en referens är ett exempel på ett prvalue (ett temporärt objekt), att annat är literaler som 4711 och **true**.

243, implementeringen av move-konstruktorn för klassen Array (det nedre kodexemplet): 1) v.p skulle också kunna kopieras i initierarlistan; 2) en generell lösning är att defaultinitiera destinationsobjektet och sedan använda swap(), vilket alla containerklasser bör ha. Det objekt som lämnar ifrån sig sina resurser måste i vilket fall vara kopierbart och tilldelningsbart - det behöver inte vara ett temporärt objekt, typiskt försätter man det i ett tillstånd som motsvarar defaultinitiering.

245, *första stycket*: Valet mellan referens och pekare för att representera relationen ”känner till” styrs också av frågan om kopiering av objekt. Väljer man referens väljer man samtidigt bort möjligheten att ha kopieringstilldelning och move-tilldelning.

245, *7.3.7 Typomvandlingskonstruktorer*: Hellre ”Typomvandlande konstruktorer”.

254, *första stycke*t: Det finns en dimension till i detta sammanhang, nämligen då det gäller kopiering av objekt av exempelvis typen Spelare. Kopiering med kopieringskonstruktor är inget problem, det är ett nytt objekt skapas och datamedlemmen leken kan då initieras som en kopia av leken i källobjektet. Kopieringstilldelning gåt däremot inte, för då ska leken i destinationsobjektet tilldelas ett nytt värde och det är inte tillåtet. Move-operationer är inte heller möjligt. Lösningen är att låta leken vara en pekare och vill man göra detta snyggt finns det en hjälpklass std::reference_wrapper som göt det möjligt att deklarera objekt som beter sig som referenser men kan kopieras och tilldelas.

8. Mer om klasser

Behandlingen av operatorer är inte bra, till exempel ges felaktiga anvisningar och motivationer för val av returtyp för vissa viktiga operatorer (tilldelning). Uppdelning i att antingen ska en operator vara en medlemsfunktion eller så ska den vara en vänfunktion är en felaktig syn. Valet sår i första hand mellan medlem eller icke-medlem; om en icke-medlem behöver vara vän är en annan och senare fråga.

Liksom tidigare råder stor förvirring kring referenser, speciellt rvalue-referenser (och vad nu ”tillfälliga uttryck” skulle kunna vara).

Pekare till medlemmar (8.7) är avancerat och det blir obegripligt när det inte tas upp i något meningsfullt sammanhang. Ett ställe där exempel med pekare till medlemsfunktioner skulle kunna dyka upp är i samband med funktionsadaptrar i 12.3.3. Det som tas upp där då det gäller Adapterfunktioner är dock både deprecated och ofullständigt.

Faktarutor kapitel 8

262, *Konstanta medlemsfunktioner*: Funktioner som deklareras **const** borde betecknas **const**-medlemsfunktioner – det inte är funktionen som är konstant, utan det innebär att det objekt som funktionen anropas för behandlas som konstant inuti funktionen (**const** är relaterat till **this**-pekaren). Det skulle också kunna påpekas att **const** måste anges i både deklarationen och i definitionen om man gör en sådan uppdelning. Kodexemplet visar en deklaration, i faktarutan sägs definition.

267, *Vänner*: Det saknas att man också kan ange att en viss medlemsfunktion i en anna klass ska vara vän, **friend** typ C2::memfun(parametrar); Funktionen memfun(parametrar), som är medlemsfunktion i klassen C2, får tillgång till alla medlemmar i klassen C, oavsett åtkomstspecifikation – att det står ”privata” medlemmar skapar ovisshet om hur det är med **protected** (även om man kan tycka att det inte borde).

269, *Operatorer*: ”vänfunktioner” bör vara ”icke-medlemsfunktioner (vän endast vid behov).

273, *Binär operator*: Det borde även anges att en binär operator ska deklareras med två parametrar om den deklareras som icke-medlem.

277, *Unär operator*: Det borde även anges att en unär operator ska deklareras med en parameter om den deklareras som icke-medlem.

278, *Operatorerna ++ och --*: Stryk **const** i returtypen för prefix **operator**++. Det kunde vara bra att påpeka att man inte ska deklarera ett namn för parametern i postfix **operator**++. Det visar tydligare att det är en ”dummyparameter” och man undviker då onödiga varningar från kompilatorn om att ”parametern x används inte”. Det är helt korrekt att inte namnge en parameter, även i en funktionsdefinition, om den inte är avsedd att användas inuti funktionen.

279, *Tilldelningsoperator*: Bör vara *Kopieringstilldelningsoperator*. måste vare en icke-statisk medlemsfunktion. Ska en klass även ha en move-konstruktor (move-konstruktor) skapas tvetydighet med en parameter som är ”objekt av samma klass”, man ska då välja ”lvalue-referens till konstant”. För klasser som inte ska ha move-operationer kan värdeparameter övervägas, men då ska det finnas en anledning. I den sista meningen skulle även ”och en egen destruktor” kunna läggas till.

282, *Indexeringsoperatorer*: Måste vare en icke-statisk medlemsfunktion. Indextypen väljs vanligtvis som std::size_t (<cstddef>, size_t är en **unsigned** typ). Normalt *ska* den överlagras i en icke-**const** och en **const**-version.

283, *Funktionsanropsoperatorn*: Måste vare en icke-statisk medlemsfunktion, bör deklareras **const** om möjligt.

286, *Vänfunktioner som operatorer*: Borde vara *Operatorer som icke-medlemmar* (liksom rubriken på hela avsnitt 8.4.8) – att en icke-medlemsfunktion kan behöva vara vän (**friend**) är en annan fråga.

289, *Typomvandlingsoperator*: Hellre *Typomvandlingsfunktion* eftersom standarden använder ”conversion function”). Det kunde anges i faktarutan att en typomvandlingsfunktion kan deklareras **explicit** (framgår av den efterföljande texten men är väsentligt).

294, *Statiska medlemsfunktioner*: Det borde påpekas att statiska medlemsfunktioner kan anropas även via objekt, x.namn(parametrar), eller via pekare till objekt, p->namn(parametrar), men även i sådana fall får naturligtvis endast statiska medlemmar användas.

Övrig text och kodexempel i kapitel 8

261, *8.1 Konstanta objekt, första stycket, första raden*: I C++ är begreppet objekt betydligt vidare än att omfatta enbart variabler och konstanter av klasstyp (en variabel av typ **int** är exempelvis ett objekt).

263, *tredje stycket från slutet*: Vilken typ **this**-pekaren har beror på hur medlemsfunktionen i fråga är deklarerad. I detta fall gäller att i **const**-funktionerna har **this** typen **const** Klocka*, i icke-**const**-funktionerna typen Klocka*. Det är detta som är effekten av att deklarera en medlemsfunktion **const** eller ej (även **volatile** kan förekomma.)

265, *8.3 Vänner, första stycket, näst sista meningen*: En vän kan också vara en viss medlemsfunktion i en annan klass, ”friend returtyp C2::memfun(parametrar)”. Det finns alltså tre varianter av vänner.

266, *första stycket*: Boken ger lätt uppfattningen att, då man ska deklarera en operation för en klass, valet står mellan medlemsfunktion eller vänfunktion. De två huvudalternativen är snarare medlem eller icke-medlem. Om en icke-medlem kan implementeras med **public**-medlemmar ska den inte göras till vän, såvida det inte finns uppenbara effektivitetsvinster som kan motivera vänskap. Vänskap är den starkaste typen av koppling som kan skapas mellan klasser och funktioner och grundregeln är därför att undvika. Det kan också tilläggas att icke-medlemsfunktioner som anses höra till en klass, vänner eller inte, bör deklareras i samma namnrymd som klassen. I annat fall kanske inte funktionerna hittas av kompilatorn/länkaren eller att en annan funktion än den avsedda hittas och väljs. Detta kallas *argument-dependent lookup*, ADL, och är en viktig egenskap hos C++. ADL innebär att då en funktion (ej medlemsfunktion) anropas avgör parametrarnas typ var funktionen kommer att sökas och i princip sker sökningen det de namnrymder där parametertyperna är deklarerade.

261, *8.1 Konstanta objekt*, första stycket: begreppet objekt får man se upp med. I sammanhanget objektorienterad programmering avser objekt det som sägs i boken, ”en variabel vars typ är en klass”, men i C++ omfattar objekt även variabler och konstanter av exempelvis grundläggande typ (**int**, **double**, etc.).

265, *8.3 Pekaren this*: tilläggas kan att den exakta typen för this beror på hur en medlemsfunktion i en klass X är deklarerad. Om varken **const** eller **volatile** är typen X*, om **const** är typen **const** X*, om **volatile** är typen **volatile** X*, om både **const** och **volatile** är typen **const volatile** X*. I en **const**-deklarerad medlemsfunktion är alltså åtkomstvägen till objektet **const** X*, dvs objektet är konstant i denna kontext och kan därför inte modifieras.

268, *8.4 Operatorer*, tredje stycket: Följande kan vara värt att lägga till. De operatorer som inte kan överlagras är punktoperatorn (.), räckviddsoperatorn (::), medlemsåtkomstoperatorn (.*) och villkorsoperatorn (?:). Att tillåta överlagring av de tre första skulle medföra stora problem, eftersom de är grundläggande för åtkomst till medlemmar. Villkorsoperatorn (x ? y : z) är problematisk på grund av den strikta beräkningsordning som gäller för dess argument (x ska beräknas först) och vad som ska vara dess resultat (om x är true ska y beräknas och vara resultatet, om x är false ska z beräknas och vara resultatet). Bland de operatorer som kan överlagras måste tilldelningsoperatorn (=), indexeringsoperatorn ([]), funktionsanropsoperatorn (()) och piloperatorn (->) vara icke-statiska medlemsfunktioner. Det säkerställer att vänsteroperanden är ett objekt av typen i fråga och inte, till exempel, ett temporärt objekt som skapats genom automatisk typomvandling.

269, fjärde stycket: de två huvudalternativen för att överlagra operatorer för klasser är inte som medlem eller vänfunktion, utan som medlem eller icke-medlem. Vänskap behöver man bara tillgripa om inte operatorn kan implementeras med enbart klassens publika gränssnitt (vänskap är den värsta formen av beroende).

270, första stycket: kunde tilläggas att den underförstådda parametern till **operator**== är objektet som **this** anger.

271, andra stycket: **operator**+= är egentligen inte en additionsoperator, utan en sammansatt tilldelningsoperator.

271, deklarationen och definitionen för MyArray::**operator**+=: returtypen ska vara MyArray&, utan **const**, eftersom en tilldelningsfunktion förväntas returnera ett lvalue (endast speciella fall skulle kunna motiverat något annat).

272, första stycket, andra raden: ”detta ska vara en kopia” är fel i sig och returtypen **const** MyArray& är inte en kopia, utan en referens till det objekt som är vänsteroperand. Returtypen ska dessutom vara MyArray& (ett *lvalue*) enligt standard och gängse konvention.

272, första stycket, andra raden: **operator**+= det är en (sammansatt) tilldelningsoperator och den ska returnera MyArray& (ett lvalue).

272, första stycket, tredje raden från slutet: Resultattypen ska *inte* anges som referens till konstant, det *ska* vara möjligt att ändra objektet som står till vänster om += (ska returnera ett lvalue).

272, exemplet mitt på sidan: Returtypen är bra men en kommentar skulle kunna vara att man normalt brukar deklarera **operator**+ som icke-medlem. I detta fall kan man också göra det utan problem eftersom operatorn är implementerad med += (som är **public**, ingen vänskap behövs).

273, deklarationen och definitionen för MyArray::**operator**+=: stryk **const** i returtyperna.

273, deklarationen för MyArray::**operator**&=: stryk **const** i returtypen.

274, definitionen för MyArray::**operator**&=: stryk **const** i returtypen.

274, **operator**&: förstahandsvalet vore att deklarera som icke-medlem.

275, *8.4.2 Fördefinierade jämförelseoperatorer*: Använd inte dessa operatorer! Definiera < och == för din klass från grunden och sedan de övriga (>, <=, >=, !=) i termer av dessa två. Användning av rel_ops innebär att namnrymden std::rel_ops måste inkluderas och öppnas i kod som använder din klass och det vill man inte göra – tillgängliggör rel_ops för alla typer som har < och ==, vilket kanske inte är önskvärt.

275, första stycket, mitt i: ’util’ ska vara ’utility’.

276, stycket före 8.4.3: Sammanfattningsvis gäller att du ska definiera de likhets- och relationsoperatorer som du vill ha helt själv.

277, deklarationen för prefix **operator**++: stryk **const** i returtypen.

277, definitionen för prefix **operator**++: stryk **const** i returtypen.

278, *8.4.4 Tilldelningsoperatorn*: borde vara Kopieringstilldelningsoperatorn. Det finns dessutom 10 sammansatta tilldelningsoperatorer motsvarande operatorerna *, /. %, +, -. <<, >>, &, ^och |.

279, deklarationen för **operator**=: stryk **const** i returtypen.

279, definitionen för **operator**=: stryk **const** i returtypen. Implementeringen är inte undantagssäker. Om new skulle misslyckas (dynamiskt minne slut eller om elementen som lagras vore av klasstyp med risken för att undantag kastas då elementen kopieras) har vi redan lämnat tillbaka minnet för det aktuella objektet. Vad som då kan hända är odefinierat, objektet är i ett odefinierat tillstånd. En hjälpvariabel kan användas för att ta emot pekaren till det minne som new erhåller och om detta gick bra är vi på den säkra

sidan och kan vid lämpligt tillfälle återlämna det gamla minnet. Det finns en teknik som kallas ”skapa en temporär/kopia och byt” som skulle göra detta mer elegant om MyArray hade haft en funktion för att byta innehåll på två MyArray-objekt (vilket varje containerliknande klass bör ha och brukar heta swap).

280, *8.4.5 Tilldelningsoperatorn, move-version*: Hellre move-tilldelningsoperator.

280, *8.4.5, första stycket*: ”Denna version fick en referens till tillfälligt uttryck som parameter” bör vara ”Denna version fick en rvalue-referens som parameter”. Parametern binds till det argument som ges i anropet och det är ett rvalue, typiskt ett temporärt objekt men inte alls givet.

280, definitionen för MyArray::**operator**=: stryk **const** i returtypen. Inget problem med **delete** i detta fall men man kan, som i fallet kopieringstilldelningsoperatorn, hitta en mer elegant implementering om klassen har en clear-funktion och en swap-funktion. Anropa clear för det aktuella objektet (”nollställ”), sedan swap för att byta innehåll. Det objekt som lämnar ifrån sig sina resurser måste gå att använda fortsättningsvis (måste gå att destruera, kopiera och tilldela), typiskt försätter man det i ett tillstånd som motsvarar defaultinitiering (”nollställning”). Förutsätt inte att objektet som resurser ska flyttas från är ett temporärt objekt.

281, *8.4.6 Indexeringsoperatorn*: I princip inget fel att även ha vanliga funktioner för åtkomst till element. När de motsvarar indexeringsoperatorn brukar de av konvention heta at() och kasta undantag om argumentet ligger utanför tillåtet intervall.

281, *andra kodexemplet (mitt på sidan)*: Detta ska ge kompileringsfel för **operator**=. Vid ”defaultning” eller ”deletening” måste deklarationen överensstämma med den deklaration en implicit deklarations skulle ha och där ingår inte **const** i returtypen (en tilldelningsoperator för en klass X *ska* returnera X&).

283: kodexemplet för MyArray::**operator**(): stryk **const** i returtypen.

285, implementeringen av **operator**<<: Ett av de sämsta variabelnamn man kan hitta är ’o’; ’os’ (output stream) skulle saken betydligt bättre.

287, andra stycket, näst sista raden: ”måste vara en fristående funktion”? Kan vara medlem men den bör vara en fristående funktion.

287, tredje stycket: ”den vänstra parametern refererar till ett tillfälligt uttryck” är nonsens - den vänstra parametern har typen rvalue-referens till MyArray. Vad parametern egentligen refererar till är inte relevant i detta sammanhang.

287, sista stycket: man bör inte likställa grund kopiering med flytt.

288, *8.4.10 Typomvandlingsoperatorer*: bör vara Typomvandlingsfunktioner (”conversion functions”).

9. Arv

Det finns några fel och lite okunniga kommentarer om designen av C++. Det i sammanhanget grundläggande begreppet *override*, *överskugga*, är helt förbigånget. override nämns men endast som ett ord som kan förekomma i kod och man ges (genom att det skrivs med fet stil) intrycket av att det skulle vara ett reserverat ord (nyckelord), vilket det inte är.

Det som sägs i *9.7 Förbättrad kontroll vid dynamisk bindning* efter det första stycket är mestadels fel, endast det som sägs om ordet override stämmer med standarden. *9.11 Virtuella operatorer* förefaller så avancerat att man kan fråga sig om det alls borde tas upp i en bok av detta slag? Det finns enklare exempel än det som visas i boken, om man vill ge exempel på operatorer som ska bete sig polymorft, till exempel **operator**<< för att skriva ut objekt som tillhör en polymorf klasshierarki.

Faktarutor kapitel 9

306, *Härledda klasser*: konstruktorer kan ärvas om man i en härledd klass namnger en konstruktor i en **using**-deklaration, **using** B::B.

311, *Konstruktorer vid arv*: Huruvida ”den egna klassens datamedlemmar initieras” beror på datamedlemmarnas typ. ”satserna innanför klamrarna” är alltså funktionskroppen.

312, *Destruktorer vid arv*: Det fattas en punkt – efter att satserna i destruktorns kropp utförts destrueras klassens egna datamedlemmar (i omvänd ordning till hur de deklareras) och sedan anropas basklassens destruktor. (Om det finns flera basklasser, multipelt arv, anropas basklassdestruktorerna i omvänd ordning till hur basklasserna deklarerats och alltså initierats).

335, *Några begrepp*: Begrepper ”override”, *överskugga* på svenska, tycks undvikas helt och dyker enbart upp som den speciella identifieraren **override** (sid 342). I stället talas om att *omdefiniera* virtuella funktioner, och det kan man göra, men att inte använda override/överskugga är märkligt, liksom att inte jämföra överskuggning och överlagring – två snarlika men ändå skilda begrepp.

340, *Dynamisk bindning*: (sista meningen) I fallet p är det snarare typen för uttrycket *p som anger den dynamiska typen.

345, *Operatorerna typeid och dynamic_cast*: kravet att inkludera typeinfo gäller enbart för typeid (som skapar objekt av typen std::type_info). Vad medlemsfunktionen name() returnerar kan variera, det är inte definierat av standarden, i vissa system kan det vara det namn som deklareras i programmet, i andra kan det vara ett (kryptiskt) internt namn som kompilatorn och länkaren använder.

348, *Virtuella destruktorer*: enbart polymorfa klasser bör ha en virtuell destruktor (det är inte givet att härledning innebär att objekt ska användas polymorft).

352, *Abstrakta klasser*: Det är fel att rena virtuella funktioner inte har någon definition, det kan de. Definitionen måste i så fall ges separat och den kan bara anropas av klassens egna eller subklassernas medlemsfunktioner. Att en subklass har en ”egen version”, dvs överskuggning, av en ren virtuell funktion kan även innebära en ärvd överskuggning.

Övrig text och kodexempel i kapitel 9

304, *stycket före det andra kodexemplet, på första raden*: ”mellan arvshierarkier” borde väl vara ”inom en arvshierarki”? Att två separata basklasser och deras respektive subklasser utgör två disjunkta deklarativa områden är uppenbart och då är inte överlagring aktuellt. Inom en klasshierarki utgör en basklass och en subklass två separat deklarationsområden och kan i grunden jämföras med att ett namn i en nästlad blocksats döljer samma namn i en omgivande blocksats.

304, *sista stycket*: Man får lätt uppfattningen att konstruktorer inte kan ärvas, men som framgår senare är det möjligt och det kunde ha påpekats till exempel med en fotnot.

306, *sista stycket*: Fenomenet som beskrivs, typomvandling av ett subklassobjekt till ett basklassobjekt, kallas ”slicing” och är inte så oproblematiskt som görs gällande. Det är i vilket fall något som man normalt undviker, till exempel genom att överföra parametrar av klasstyp som referens till konstant i ställer för genom kopiering.

307, *det första kodexemplet (övre delen av sidan)*: Ingen kompetent programmerare skulle väl skriva en sådan funktion, parametrarna ska vara ”**const** Anställd&”. Varför lära ut dålig kodningsstil?

307, *kodexemplet nedtill på sidan*: De fyra typomvandlingsoperatorerna **static_cast**, **dynamic_cast**, **reinterpret_cast** och **const_cast** har införts i C++ av goda skäl och ska användas. Programmeraren visar genom valet av någon av dessa fyra sin avsikt och, även om det inte är idiotsäkert, minskar risken att göra tokiga omvandlingar. Samliga fyra har sina begränsningar då det gäller användning och väljer man en olämplig form får man veta det. I detta exempel bör **dynamic_cast** användas.

312, *9.3 Destruktorer vid arv, första stycket*: Lite missledande att skriva ”En destruktor används normalt bara i de fall som en konstruktion i klassen har allokerat minne som behöver frigöras”. Varje subobjekts destruktor körs alltid och alla datamedlemmar som har en destruktor får i samband med detta sin destruktor körd (saknas i den följande faktarutan). Däremot är det riktigt att återlämning av dynamisk minne är en vanlig uppgift, vare sig klassen ifråga uttryckligen hanterar sådant eller har datamedlemmar som gör det (men det behöver inte objektet ifråga bekymra sig om).

313, *exemplet som demonstrerar utförandet av destruktorer*: Det hade varit lite mer illustrativt vad som sker om klass C1 och C2 även hade haft var sin datamedlem av exempelvis typ string, s1 respektive s2, deklarerade till exempel sist i medlemslistan. I konstruktorerna hade då defaultkonstruktorn för dessa körts (underförstått) sist i medlemsinitierarlistorna och i destruktorerna hade destruktorerna får s1 och s2 körts efter att satserna i respektive destruktors kropp körts, innan basklassens destruktor anropas.

317, näst sista stycket: Varför skulle man göra något sådant?!

318, *stycket före 9.5*: Det är stämmer inte att man inte kan göra en medlem mer synlig än den är basklassen, det gäller enbart **private**-medlemmar. Det är tvärt om en typisk användning av **using**-deklarationen. Påståendet i stycket under det andra kodexemplet på sidan 318, ”t.ex. hade försökt lägga användningsdeklarationen av 12 i public-delen av D5 så hade vi fått ett fel” är fel. I denna kontext är i2 protected, det kan inget ändra, speciellt inte den tillgänglighetsspecifikation som anges i deklarationen av D5, den har relevans först för subklasser till D5. Att lyfta en **protected**-medlem till **public**-delen innebär egentligen enbart att en medlem som är tillgänglig, i och för sig indirekt, görs direkt tillgänglig, vilket man naturligtvis enbart gör om det är vettigt.

319, *klassen MyArray*: **const** ska tas bort i returtypen för alla tilldelningsoperatorer (7 st), prefix **operator**++, prefix **operator**-- och funktionsanropsoperatorn. Man kan även fundera över vilka av de **friend**-deklarerade funktionerna som egentligen behöver vara vänner.

341, *9.7 Förbättrad kontroll vid dynamisk bindning*: Det som påstås stämmer i stora delar inte.

342, första kodexemplet: ’explicit’ kan inte förekomma i deklarationen av en klass, däremot ’final’ som då innebär att klassen inte kan härledas från (inga subklasser kan deklareras). Observera att ’final’ inte är ett reserverat ord, utan en vanlig identifierare som får speciell mening i denna användning. Detsamma gäller ’override’ när den identifieraren används för att märka en virtuell medlemsfunktion på det sätt som görs i exemplet.

342, *tredje stycket*: Om man märker en (virtuell) funktion med ’override’ kontrollerar kompilatorn att en motsvarande virtuell funktion har deklarerats i någon basklass. Det som sägs för övrigt är fel.

342, *fjärde stycket och till sidans slut*: Det som sägs är helt fel (**new** kan inte användas som anges).

10. Exceptionella händelser

Det brister en del då det gäller uppdatering med avseende på C++11, bland annat har det införts ett antal standardundantag vid sidan av de som anges i figuren på sidan 363.

Faktarutor kapitel 10

365, *Throw-sats*: syntaktiskt är det ett *uttryck*, **throw**-uttryck, vilket typiskt används som en sats, uttrycks-sats, genom att avsluta med ett semikolon.

374, *Parametrar till hanterare*: Den första varianten, ”catch by copy”, ska man nog helst undvika Om typen för det kastade undantaget är en subtyp till typen deklarerad i **catch** sker ”slicing” vid kopieringen. Den andra varianten, ”catch by reference”, kan deklareras **const** T&, bör föredras. Ingen kopiering sker, ingen slicing kan inträffa, och anrop av virtuella medlemsfunktioner, som what(), binds dynamiskt.

Övrig text och kodexempel i kapitel 10

363, figuren upptill: En nytt, femte undantag under **logic_error** är **future_error** (har med trådar att göra). Under **range_error** har **system_error** tillkommit och under det hittar man numera **ios_base::failure**, som tidigare var en direkt subklass till **exception**.
Andra direkta subklasser till **exception** är **bad_typeid**, **bad_cast**, **bad_weak_ptr**, **bad_function_call**, **bad_alloc**, och under det **bad_array_new_length**, och **bad_exception**.

363, exemplet som visar hur en undantagsklass kan härledas: dels ska konstruktorer i undantagsklasser deklareras **explicit**, dels har man i C++11 även en konstruktor som har **const char*** som parametertyp.

```
class klassnamn : public basklass {
public:
   explicit klassnamn(const string& what_arg);
   explicit klassnamn(const char* what_arg);
};
```

Motsvarande tillägg ska göras för de tre undantagsklasserna **io_error**, **end_error** och **data_error** på sidan **376**.

370, kodexemplet: Det finns en regel som säger ”*throw by value, catch by reference*”. Det första syftar på att man *inte* ska göra **throw new** range_error("Nej!"), dvs *inte* kasta pekare till ett dynamisk allokerat objekt (vem ska göra **delete**?), utan man ska kasta ett temporärt objekt, **throw** range_error("Nej!"). Det senare syftar på att man ska fånga som **catch** (**const** exception& e), dvs så att det inte skapas en kopia i parametern (e).

379, *Specifikation av exceptionella händelser*: Allt som sägs fram till det allra sista stycket i detta avsnitt behandlar så kallade *dynamic exception specification*, **throw**(…), vilket är *deprecated*. I C++11 har detta ersatts av noexcept specification, **noexcept**. Redan tidigare var konventionen att endast använda throw(), dvs endast specificera då en funktion inte kastar undantag, vilket motsvarar **noexcept** eller **noexcept**(**true**), och avsaknad av sådan specifikation motsvarar **noexcept**(**false**).

11. Strömmar och filer

401, kodexemplet: Filströmmar accepterar filnamn i form av riktiga strängar (std::string). Variabeln name bör deklareras som string, i stället för **char**[100], och filnamnet kan läsas enklare, cin >> name;.

411, **for**-satsen på rad 7: egentligen borde man använda cbegin() och cend() med tanke på att iteratorn cit är en const_iterator. Eftersom v är en icke-const vector returnerar begin() och end() iteratorer av typen iterator och då gör automatisk typomvandling då cit initiera och då cit jämförs med v.end().

411, **for**-satsen på rad 12: ett alternativt sätt som mer motsvarar iteratorexemplet vore att deklarera y som "**const auto**& y", vilket skulle undvika kopiering av varje element och samtidigt förhindra modifiering.

411, stycket mitt på sidan: "De iteratorer som är definierade i containerklasserna har alla egenskapen att? stega baklänges". Detta gäller inte för forward_list eller de oordnade associativa containrarna, de har inga bakåtiteratorer.

411, näst sista stycket: Det som står om var bakåtiteratorer pekar stämmer med hur det fungerar praktiskt, vilket ofta kan räcka, men det stämmer inte med hur det egentligen är. rbegin() pekar faktiskt inte på det sista elementet utan ger samma resultat som end(), dvs representerar "förbi-slutet", och rend() anger inte någon "före-första"-iterator utan ger samma resultat som begin(), dvs pekar på det första elementet. Detta låter konstigt men att det fungerar som man förväntar beror på att när man avrefererar en baklängesiterator är det elementet i positionen innan som erhålls. Anledningen till detta, till synes märkliga förhållande är att det inte säkert existerar någon "före-första"-position i alla sammanhang. Om man tittar närmare på iteratorer eller ska konstruera egna kan man behöva vara medveten om detta.

12. Containerklasser och algoritmbibliotek

Uppdateringen med avseende på C++1 har en hel del i övrigt att önska; en del saknas, en del av det som tas upp är *deprecated*, en del som påstås är direkt fel och en del viktiga nyheter/ändringar saknas.

Faktarutor kapitel 12

442, *Iteratorer*: De fyra iteratortyper som anges i den första rutan finns inte för alla containrar, forward_list och de oordnade associativa containrarna har enbar iterator och const_iterator och dessa tillhör kategorin forward iterator.
I den andra rutan saknas cbegin(), cend() crbegin() och crend(). De returnerar alltid const_iterator eller const_reverse_iterator, oavsett om containern är **const** eller inte, medan varianterna utan 'c' i namnet endast returnerar **const**-iterator om containern är ett **const**-objekt.

I en egen ruta skull följande två iteratoroperationer kunna läggas till:
advance(*it*, n): stegar iteratorn *it n* steg; ett negativt *n* är endast tillåtet för iteratorer i kategorierna *bidirectional* och *random access* (motsvarar *it+n* för random-access-iteratorer).
distance(*first*, *last*): returnerar (*last-first*) för iteratorer i kategorin *random access*, annars det antal steg som krävs för komma från *first* till *last*; *last* måste vara nåbar från *first*.

443, *Iteratorer (forts.)*: funktionerna advance() och distance() finns också för en del iteratorer; advance(it, n) stegar fram iteratorn it n steg, motsvarar it+n för en random access-iterator.

446, *Sekvenser: Operationer med iteratorer som parametrar*: Saknas s.emplace(*i*, *arg*), där *arg* är en följd av värden som motsvarar parametrarna till en konstruktor för den typ av objekt som lagras i sekvenscontainern. Ett objekt som initieras med de angivna argumentet sätts in före positionen *i*.

Syftet med emplace-funktioner är att undvika kopior som annars kan komma att skapas vid insättning i en container.

457, *Funktionsobjekt*: Kategoriseringen avser vilka slags funktionsobjekt som används av andra komponenter i standardbiblioteket. Generellt sett kan funktionsobjekt ha godtyckligt antal parametrar och det enda kravet i så fall är egentligen att **operator**() ska vara överlagrad för klasstypen ifråga.

460, *Fördefinierade funktionsobjek*t: Saknas tre: bit_and (motsvarar bitvis och, |), bit_or (motsvarar bitvis eller, &) och bit_xor (motsvarar bitvis exklusivt-eller, ^).

462, *Adapterfunktioner*: bind1st(), bind2nd() och ptr_fun() är *deprecated* i C++11, dvs ska inte användas i nyskriven kod. Dessutom saknas två funktioner besläktade med prt_fun(), nämligen mem_fun_ref() och mem_fun(), som också *deprecated*. I den nya standarden ersätter dessa gamla hjälpfunktioner av bind() och mem_fn() samt mallklassen function. Det är egentligen endast not1() och not2() som kräver att argumentet är ett funktionsobjekt och dessutom med vissa medlemstyper, de övriga kan ges argument som är vanliga funktioner och ptr_fun() är till för vanliga funktioner.

Övrig text och kodexempel i kapitel 12

441, sista stycket: att ++ stegar en baklängesiterator baklänges är egentligen ingen "underlighet". Om det inte vore så skulle, till exempel, alla algoritmer behöva finnas i två versioner, en för (framåtiteratorer (som stegas med ++) och en för bakåtiteratorer (som då skulle stegas med --).

444, stycket mitt på sidan: De nya oordnade associativa containrarna saknas; dessa har iteratorer i kategorin *forward iterator*.

446, *12.1.3 Iteratorer och strömmar* (*Strömiteratorer* hade varit en annan tänkbar rubrik): Det saknas en viktigt information, nämligen att istream_iterator använder **operator**>> för att läsa värden och att ostream_iterator använder **operator**<< för att skriva värden. Att dessa finns för den typ av värden som ska läsas/skrivas är alltså en förutsättning för att kunna använda strömiteratorerna.

447, första stycket efter 12.2: ”En fullständig uppräkning av alla algoritmerna i C++ standardbibliotek ges i appendix på sidan 615.” är fel, det saknas de cirka 20 som lagts till i C++11.

448, från sista meningen i det andra stycket och framåt: back_inserter, front_inserter och inserter är *inte* iteratortyper, utan hjälpfunktioner för att skapa motsvarande insättariteratorer (back_insert_iterator, front_insert_iterator respektive insert_iterator). Utan hjälpfunktionen back_inserter skulle copy-exemplet skrivas:

```
copy(v.begin(), v.end(), back_insert_iterator<deque<int>>(e));
```

Motsvarande för hjälpfunktionen front_inserter och iteratortypen front_insert_iterator, och för hjälpfunktionen inserter och iteratortypen insert_iterator.

454, klassen Kvaderare: välj **struct**!

460, sista stycket, fram till 12.3.4 Lambda-uttryck: Det mesta som beskrivs här är föråldrat. bind1st, bind2nd och ptr_fun är deprecated och ersatta med nytt. För att binda argument finns en ny hjälpfunktion bind(), som inte är begränsad till ett eller två argument och som även i andra avseenden har ökat flexibiliteten. Exemplet på sid 461, rad 7 skrivs med bind() enligt följande:

```
bind(greater_equal<double)(), _1, 8)
```

bind() returnerar ett unärt funktionsobjekt som kan anropas med ett argument och som i sin tur anropar greater_equal med detta argument som första argument och 8 som andra argument. Platshållaren _1, som definieras i standardbiblioteket, refererar till det argument som funktionsobjektet anropas med. Platshållarna, som är _1, _2, …, osv., definieras i std::placeholders.

Det tredje kodexemplet på sid 461 skrivs på följande sätt med bind():

```
cout << count_if(v1.begin(), v1.end(), bind(greater_equal<double>(), _1, 8));
```

Det fjärde kodexemplet på sid 461 skrivs på följande sätt med bind():

```
cout << count_if(v1.begin(), v1.end(), bind(greater_equal<double>(), 8, _1));
```

Det sista kodexemplet på sid 461 är lite mer komplicerat. not1() kräver att det funktionsobjekt som ges till not1() har nästlade typdefinitioner för returtyp och argumenttyp, som heter result_type respektive argument_type. bind() lägger inte till dessa med det gör klassmallen function:

```
cout << *find_if(v1.begin(), v1.end(),
                 not1(function<bool(double)>(bind(less<double>(), _1, 8))));
```

bind() binder det andra argumentet till det binära funktionsobjektet less<**double**>() till 8, och returnerar ett unärt funktionsobjekt med signaturen **bool**(**double**). Klassmallen function instansieras för signaturen ”**bool**(**double**)” och ett temporärt objekt som initieras med det funktionsobjekt som bind() returnerar skapas. Detta temporära objekt är ett funktionsobjekt med signaturen ”**bool**(**double**)” och nästlade typdefinitioner ”**typedef bool** result_type;” och ”**typedef double** argument_type;”. Detta funktionsobjekt ges till not1(), som returnerar ett funktionsobjekt med signaturen ”**bool**(**double**)”, vilket anropas för varje element i v1 (motsvarar alltså platshållaren _1 som gavs som argument till bind()).

Om man ger en funktionspekare eller en medlemsfunktionspekare till bind(), kommer det funktionsobjekt som bind() returnerar att ha en medlemstyp result_type, som är returtypen för funktionen eller medlemsfunktionen. Om man ger ett funktionsobjekt med en nästlad typdefinition result_type till bind() kommer det funktionsobjekt som bind() returnerar att ha motsvarande medlem result_type. I annat fall kommer ingen medlem result_type att vara definierad.

En instans av function kommer alltid ha en medlemstyp result_type definierad. För en unär funktion kommer medlemstypen argument_type att vara definierad. För en binär funktion kommer medlemstyperna first_argument_type och second_argument_type att vara definierade.

462, kodexemplen med ptr_fun: ptr_fun() är *deprecated*. Syftet med ptr_fun() är att skapa ett funktionsobjekt för en vanlig funktion med typdefinitioner för funktionens resultattyp och argumenttyper, vilket vissa komponenter kräver, till exempel hjälpfunktionerna not1() och not2(). Detta kan man i C++11 göra med klassmallen function:

```
function<bool(double, double)>(nastan_lika)
```

Exemplet med find_if kan dock skrivas med enbart bind, find_if kräver inga sådana typdefinitioner; observera att det ska vara v1.end(), inte v2.end():

```
cout << *find_if(v1.begin(), v1.end(), bind(nastan_lika, _1, 6));
```

464, fjärde stycket: icke-lokala variabler och statiska lokala variabler kan användas utan motsvarande *lambda capture* i *lambda introducer*, [].

464, rad 14 och rad 11 nedifrån: ("catch" hör snarare ihop med exception-hantering) "catch by value" bör vara "capture by copy" och "catch by reference" bör vara "capture by reference", för att använda konventionella beteckningar.

464, sista stycket, rad 1: "alla" innebär alla namn som *används* i lambda-uttrycket.

465, tillägg till 12.3.4 Lambda-uttryck: [**this**] används om man ska fånga klassmedlemmar.

```
class X {
private:
   void memfun() {auto f = [this](int x) { return x < m; }; } // [m] fungerar inte
private:
   int m;
};
```

13. Dynamiska datastrukturer

Detta hade varit ett lämpligt kapitel för att introducera och motivera användning av **struct** i stället för **class** i vissa fall. Det finns flera exempel där **struct** är det självklara valet i stället för **class** (Element och Nod).

Det finns en hel del förekomster av 0 (**int**-nollan som kan typomvandlas till ett tompekarvärde) där den nya pekarliteralen **nullptr** bör användas. Se avsnittet Syntax sist.

Det gamla sättet att specificera om funktioner kan kasta undantag eller ej, så kallad d*ynamisk exception-specifikation*, **throw**(…), används på många ställen för att specificera vilket undantag som kan kastas. Det finns flera invändningar mot detta, dels är denna form av undantagsspecifikation *deprecated* och bör alltså undvikas, dels är den konvention som utvecklats att man endast bör specificera vilka funktioner som *inte* kastar undantag och att detta då ska görs med den nya formen *noexcept-specifikation*, **noexcept**.

Felaktiga och potentiellt katastrofala sätt att eliminera kopieringskonstruktor och kopieringstilldelnings-operator i flera klasser (Stack och Queue).

Det träd som konstrueras i avsnitt *13.4.3 En trädklass* och som sägs vara mer ”C++-mässigt” än de träd som konstruerats tidigare i avsnitt 13.4 framstår snarare som en akademisk syn på hur träd kan konstrueras baserad på deras rekursiva egenskaper (ett subträd är också ett träd). Den realistiska lösningen är att ha en trädstruktur bestående av enbart Noder och ett objekt av typen Tree som lagrar pekaren till ett sådant träds rotnod,. Så ska en containerklass, vilket detta är, konstrueras.

Text och kodexempel i kapitel 13

496, definitionen av Element: Alla medlemmar är synliga – typexempel på när **struct** väljs.

496, kodexemplet fyra rader från slutet på sida: Element* forsta = **nullptr**;

502, definitionen av Element: Välj **struct**.

506, definitionen av Stack: *Dynamisk exceptionspecifikation*, som **throw**(length_error), är *deprecated* och ska inte användas. Använd i stället *noexcept-specifikation*, **noexcept**(**false**) om en funktion kan kasta undantag eller **noexcept** alternativt **noexcept**(**true**) om en funktion inte kastar undantag.

Det vanliga är att *inte* specificera något alls för funktioner som kan kasta undantag, **noexcept** för funktioner som *inte* kastar undantag. Anges inget ska man alltså förvänta sig att undantag kan kastas.

506, definitionen av Stack: Helt fel att *definiera* kopieringskonstruktorn och kopieringstilldelnings-operatorn, även om de deklareras **private** (sid 507). Definitionerna gör dem anropbara och om så skulle ske av misstag innebär det katastrof. I C++11 eliminerar man dem enligt följande:

```
Stack(const Stack&) = delete;
Stack& operator=(const Stack&) = delete;
```

(I C++98 *deklarera* man dem enbart och avsaknaden av *definition* ger då kompileringsfel, om de råkar anropas av misstag. Bokens lösning ger, i bästa fall, exekveringsfel.)

508, kodexemplen: **throw**(length_error), är *deprecated* (ta bort).

510, kodexemplen: **throw**(length_error), är *deprecated* (ta bort).

511, definitionen av Queue innehåller fatala fel: Se motsvarande kommentarer till definitionen av klassen Stack på sid 506.

511-512, kodexemplen: **throw**(out_of_range) är *deprecated* (ta bort).

513, definitionen av Element: välj **struct**.

520-21, kodexemplen: **throw**(length_error) är *deprecated* (ta bort).

524, definitionen av Nod: Välj **struct**; defaultargumenten för v och h bör vara **nullptr**.

530, definitionen av Nod: Välj **struct**; defaultargumenten för v och h bör vara **nullptr**.

530: Defaultargumentet 0 för parametern n (för namn) ger typiskt "segmentation fault" i strcpy(). Tompekare är inte tillåtet som argument till strcmp men strcmp kontrollerar inte detta; programmeraren förutsätts se till att det inte blir fel men här ber man ju om det… Defaultargumentet bör tas bort eller så använder man std:.string i stället för namn.

531, *13.4.3 En trädklass*: Förefaller vara en tämligen orealistisk implementering av en trädklass (”akademisk”). Mer realistiskt om själva trädet består av en typ an noder, i stil med de trädnodklasser som används i 13.4.1 och 13.4.2 och att klassen Trad har ett träd bestående av sådana noder som intern representation, vilket skulle göra Trad till en containerlikande klass.

532, trädet i figur 13.18: Helt orealistiskt…

533, kodexemplen: **throw**(range_error), är *deprecated* (ta bort).

14. Mallar

”Variadic templates” (nytt i C++11) är inte med och kanske kan ses som väl avancerat i en bok av detta slag, men borde ha nämnts. Det finns några brister då det gäller kompletthet och annat av principiell natur. För övrigt ofullständigheter och brister i kodexemplen.

Det finns en hel del förekomster av 0 (**int**-nollan som kan typomvandlas till ett tompekarvärde) där den nya pekarliteralen **nullptr** bör användas. Se avsnittet Syntax sist.

Faktarutor i kapitel 14

549, *Mallparametrar*: Det finns en tredje form av parameter, *mallparameter*, som alltså har formen **template**<parametrar> class T. Man kan alltså använda en mall för att instansiera en annan mall.

En stor och viktig nyhet i C++11 är att man kan definiera mallar som kan ta ett varierande antal instansieringsargument, *variadic templates*. När man deklareras en sådan mall används ellips-notation, till exempel **class**… P eller **typename**… P, vilket kallas *parameter pack*. En mall med ett sådant *parameter pack* kan instansieras med ingen, en eller flera typer.

565, *Funktionsmallar*: ”specialinstans” kan möjligtvis leda tankarna till att detta skulle vara någon form av *instans* av funktionsmallen och möjligtvis det som kallas *specialicering* av en mall, men det en vanlig funktion överlagras med mallfunktionen.

Övrig text och kodexempel i kapitel 14

540, kodexemplet Matris: Att implementera en kopieringskonstruktor med klassens egen kopierings-tilldelningsoperator är inget man vill göra. Däremot kan man tänka sig att göra tvärt om (idiomet "skapa en temporär och byt"). Kopieringskonstruktorn för Matris kan implementeras enligt nedan, rakt på sak och utan konstigheter, om man byter deklarationsordning för datamedlemmarna så de deklareras i ordning r, k och a, och dessutom utnyttjar standardalgoritmen copy (annars har man samma **for**-sats som i boken):

```
Matris::Matris(const Matris& other)
  : r(other.r), k(other.k), a(new double[r*k])
{
   std::copy(other.a, other.a + (r*k), a);
}
```

540, kodexemplet Matris: **operator**() behövs egentligen i två versionen, en **const**-version också. Det behovet inser man om man ska kunna deklarera funktionen add() på sidan 545 på ett korrekt sätt.

541, kodexemplet, **operator**=: det är ett risktagande att göra **delete** innan man erhållit nytt minne; skulle **new** misslyckas (kastar bad_alloc) har man försatt objektet som ska tilldelas i ett irreparabelt, otillåtet tillstånd (**operator**= är därmed inte undantagssäker).

542, kodexemplet: ant_rad och ant_kol ska vara **const**.

541, kodexemplet, **operator**=: det är ett risktagande att göra **delete** innan man erhållit nytt minne (se 541).

544, fotnot 1: Man kan dela upp på traditionellt sätt och anpassa till olika kompilatorers konventioner genom preprocessorteknik. Inkluderingsfil (*.h) som vanligt och separata deklarationer på separat fil, som typiskt ges suffixet .icc (”include cc”). Inkluderingsfilen kan i slutet inkludera icc-filen, vilket kan göras villkorligt för att öppna för olika kompilatorlösningar. icc-filen har typiskt också en inkluderings-gard, precis som vanliga inkluderingsfiler alltid ska ha.

545, kodexemplet add(): Om man ska deklarera add() korrekt ska parametrarna ska vara **const** Matris<**int**>& och då upptäcker man att **+=** också måste finnas i en **const**-version.

547, klassen Stack: som konstaterades på sid 506 innehåller definitionen av stack fatala fel.

548, första stycket i avsnitt 14.1.4 Mallparametrar: Det finns tre kategorier av mallparametrar, *typ-parametrar*, *icke-typparametrar* (hellre än värdeparametrar) och *mallparametrar* (parametern är en mall).

550, första kodexemplet: Klassen Node bör vara **struct**, **friend** behövs då inte. I konstruktorn finns en potentiell minneskäcka, om andra **new** misslyckas, kommer det minne som första **new** redan erhållit att tappas bort.

551, kodexemplet Person: Välj **struct**.

553, kodexemplet mitt på sidan: skriv() är inte realistisk att ha som egenskap.

554, kodexemplet mitt på sidan: skriv() är inte realistisk att ha som egenskap.

556, klassen List: Som sägs är klassen förenklad. Följande kan vara värt att påpeka om det. För att man ska kunna operera även på List-objekt som är **const** måste Iterator finnas i en iterator-till-konstant-version och flera av List-funktionerna måste finnas i både en **const**- och en icke-**const**-version, till exempel:

```
const T& front() const;
const T& back() const;
```

Överför man ett List-objekt till en funktion som ”**const** List<T>&” uppstår detta behov.

~List bör för övrigt inte ”gömmas” på slutet, utan deklareras tillsammans med konstruktorerna.

557, kodexemplen: front() och back() ska finnas i **const**-varianter:

```
const T& List<T>::front() const { samma implementering som icke-const }
const T& List<T>::back() const { samma implementering som icke-const }
```

562, min-funktionerna: kodningsstil – ett alternativt sätt att implementera dessa som generellt kan anses bättre, genom att det finns en **return**-sats som otvetydigt kommer att utföras om ingen annan:

```
if (a < b)
   return a;
return b;
```

563, kodexempel swap: läge för move-semantik – både a och be ska erhålla ett nytt värde och temp är en hjälpvariabel (standardbibliotekets motsvarande swap-funktion är implementerad på detta sätt).

```
template<typename T>
void swap(T& a, T& b)
{
   T temp = std::move(a);
   a = std::move(b);
   b = std::move(temp);
}
```

564, övre kodexemplet: Ett ännu bättre alternativ är:

```
template<typename T, size_t N>
T& min_element(T (&a)[N])
{
   int m = 0;
   for (int i = 1; i < N; ++i)
      if (a[i] < a[m])
         m = i;
   return a[m];
}
```

564, nedre kodexemplet: se 564 ovan.

570, kodexempel abs_val: unary_function är *deprecated* (använd inte); deklarera i stället följande nästlade typer i abs_val:

```
typedef T result_type;
typedef T argument_type;
```

570, kodexempel abs_val: binary_function är *deprecated* (använd inte), deklarera i stället följande nästlade typer i close:

```
typedef bool result_type;
typedef T first_argument_type;
typedef T second_argument_type;
```

572, 14.3.1 *Den klassiska tekniken*: Fordon-objekt skapas med **new** och om man ska ta bort sådana ur en lista måste någon göra **delete**. Då måste Fordon ha en virtuell destruktor och någon måste göra **delete**. List::ta_bort_forsta(), till exempel, måste hålla i den första noden då den länkas förbi och sedan göra **delete** på den. För att destrueringen ska göras korrekt måste ListElement då ha en virtuell destruktor. Bokens lösning skulle fungera i ett språk med garanterad ”garbage collection”.

572, kodexemplet ListElement: Virtuell destruktor måste finnas (**noexcept** bör vara med):

```
virtual ~ListElement() noexcept = default;
```

Datamedlemmen nasta skulle kunna initieras så här (och då behöver inte defaultkonstruktorn deklareras)

```
ListElement* nasta = nullptr;
```

573, kodexempel Lista::lagg_sist: En kanske något mer läsbar implementering av **else**-grenen kan vara:

```
ListElement* p = forsta;
while (p != nullptr)
   p = p->nasta;
```

577, klassen Formel (tre kodexempel): funktionen berakna() ska vara **const**. Destruktorn kan deklareras:

```
virtual ~Formel() noexcept = default;
```

578, klassen Cell: funktionen definierad() ska vara **const**.

579, exception-klassen kalkylfel: konstruktorn bör vara **explicit**. Det ska också finnas en string-variant:

```
explicit kalkylfel(const string& what_arg = "") : logic_error(what_arg) {}
```

580, klassen Annan: funktionen berakna() ska vara **const**.

581, klassen Ark: funktionerna ant_rad() och ant_kol() ska vara **const**.

580, klassen S1: funktionen skriv() ska vara **const**.

580, klassen S2: funktionen kvad() ska vara **const**.

15. De sista pusselbitarna

594, *15.3 struct*: Det är fel att behandla denna datatyp först i detta avslutande ”slaskkapitel”. **struct** ska används för klasstyper med enbart synliga (**public**) medlemmar och hade, om inte tidigare, varit lämpligt att introducera redan i kapitel *13. Dynamiska datastrukturer*, där flera sådana klasstyper används.

För övrigt inga synpunkter på innehåll eller annat i detta kapitel.

Appendix

Appendix A

De ord som är listade i appendix A kallas egentligen *keywords*, en striktare kategori av reserverade ord, *reserved words*. Ett *keyword* är oreserverat reserverat (utom då det används som så kallat *attribute token*), medan ett *reserved ord* endast är reserverade i vissa sammanhang (main, till exempel, är endast reserverat som funktionsnamn i global namnrymd; medlemsfunktioner, klasser och uppräknare kan kallas main, men man bör förstås göra det med eftertanke).

Det fattas två ord i sammanställningen, **alignas** och **decltype**. Dessutom är **register** ”deprecated”, dvs kan med tiden komma att tas bort helt och ska därför inte användas i nyskriven kod, och **export** är borttaget men reserverat för framtida bruk.

De två identifierarna *final* och *override* kunde ha nämnts i detta sammanhang. De är inte reserverade ord men de har en speciell betydelse i vissa sammanhang.

Appendix C

Appendix C är inte uppdaterat för C++11. Ett tjugotal algoritmer har tillkommit, bland annat på grund av att move-semantik har införts och att std::initializer_list tillkommit. Följande algoritmer har tillkommit i C++11.

C1. *Söka:* find_if_not, any_of, all_of, none_of, max och min för initializer_list, minmax och minmax_element.

C3. *Kopiera och flytta element:* copy_n, copy_if, move (som copy men element flyttas om flyttilldelning finns), move_backward (som copy_backward men element flyttas, om move-tilldelning finns).

C6. *Sortera:* is_sorted, is_sorted_until. swap tillhör numera <utility>.

C7. *Operationer på sorterade datasamlingar:* is_permutation, is_partitioned, partition_copy, partition_point, shuffle.

C9. *Numeriska algoritmer:* iota.

C10. *Heap-operationer:* is_heap, is_heap_until.

Syntax

Semikolonfel

Semikolon efter en funktionsdefinitioner finns på många ställen. De utgör en tom sats som inte har något med funktion att göra, enbart skräpar ner i koden och kan ge kompileringfel om man kompilerar med strikta krav på syntax. Sådana felaktiga semikolon finns på följande ställen och ska tas bort:

230, klassen C, längst ner på sidan (en förekomst)

235, klassen Flight, högst upp på sidan (en förekomst)

253, klassen Spelare, mitt på sidan (en förekomst)

302, klassen Anstalld, mitt på sidan (en förekomst)

308, klassen Person, mitt på sidan (två förekomster)

308, klassen Anstalld, längst ner på sidan (en förekomst)

315, klassen Vackarklocka, övre delen av sidan (en förekomst)

348, klassen Person, högst upp på sidan (en förekomst)

350, klassen Spelare, strax under mitten på sidan (en förekomst)

506, klassen Stack, längst ner på sidan (en förekomst)

507, klassen Stack, högst upp på sidan (två förekomster)

510, klassen Queue, längst på sidan, först förekomsten (av tre)

510, klassen Queue, nedtill på sidan, andra och tredje förekomsten (av tre): *Definiera* inte kopieringskonstruktorn och kopieringstilldelningsoperatorn om avsikten är att eliminera dem. Att definiera dem att inte något är uppenbart fel – de blir anropbara men gör inget vettigt. Kopieringskonstruktorn kommer att skapa ett objekt med ett odefinierad tillstånd. I C++11 används **delete** för att eliminera dem:

```
Queue(const Queue&) = delete;
Queue& operator=(const Queue&) = delete;
```

512, klassen Set, längst ner på sidan (en förekomst)

513, klassen Set (forts.), högst upp på sidan (en förekomst)

520, klassen Set, mitt på sidan (två förekomster)

533, klassen Trad, mitt på sidan (åtta förekomster)

547, klassen Stack, mitt på sidan (tre förekomster)

549, klassen Stack, högst upp på sidan (en förekomst)

555, klassen Element, strax under mitten (en förekomst)

556, klassen Iterator, högst upp på sidan (två förekomster)

572, klassen ListElement, övre delen av sidan (en förekomst)

572, klassen Lista, strax under mitten på sidan (en förekomst)

577, klassen Formel, högst upp (en förekomst)

596, klassen S1, mitt på sidan (en förekomst)

Tompekare – nullptr

511: 0 i push() bör vara **nullptr**.

515: 0 i sub() bör vara **nullptr**.

515: 0 i kopia() bör vara **nullptr**.

517: 0 i uni() bör vara **nullptr**.

518: 0 i snitt() bör vara **nullptr**.

524: 0 i Nod::Nod() bör vara **nullptr** (2 ggr).

530: 0 i Nod::Nod() bör vara **nullptr** (2 ggr).

530, kodexemplet längst ner: 0 i sok() bör vara **nullptr**.

540, kodexemplet: 0 i a(0) bör vara **nullptr**.

542, kodexemplet: 0 i a(0) bör vara **nullptr**.

547, det första kodexemplet: 0 bör vara **nullptr**.

551, kodexemplet högst upp: 0 bör vara **nullptr**.

572, det andra kodexemplet: 0 bör vara **nullptr**.

556, kodexemplen: 0 (två förekomster)bör vara **nullptr**.

572, kodexemplen: 0 bör vara **nullptr**.

578, kodexempel Cell: 0 bör vara **nullptr**.