C++ för dig som kan Java

Tommy Olsson, Linköpings universitet, Institutionen för datavetenskap © 2001

C++ och Java kan ytligt sett verka lika. Det är dock främst Javas syntax som är lånad från C++ och där upphör egentligen likheterna. Java har egentligen betydligt mer gemensamt med språk som Smalltalk, Lisp och Ada 95. De skillnader som finns mellan Java och C++ innebär vanligtvis påtagliga förbättringar och förenklingar i Java jämfört med C++.

C++ är ett generellt programspråk med tonvikten lagd på systemprogrammering. Språket kan sägas vara en förbättring av C, med bra stöd för dataabstraktion och stöd för procedurell, objektorienterad och generisk programmering.

Kort C++-historik

C++ historik går tillbaka till senare delen av 70-talet. Språkets upphovsman, Bjarne Stroustrup, arbetade då med sin doktorsavhandling vid universitetet i Cambridge i England. Arbetet gällde organisation av programvara i distribuerade system och Stroustrup använde en simulator i Simula för att studera detta. Simula var ett i många avseenden utmärkt språk men problem uppstod med den tillgängliga implementeringen av Simula och programmens prestanda. För att kunna fullfölja projektet tvingades Stroustrup skriva om simulatorn i systemprogrammeringsspråket BCPL (föregångare till C).

När Stroustrup var klar med sin doktorsavhandling 1979, började han på Bell Laboratories, USA, där han studerade hur UNIX-kärnan skulle kunna distribueras i ett lokalt nätverk. Återigen uppstod behov av att kunna beskriva systemmoduler och kommunikationen mellan dessa. Stroustrup hade funderat mycket på detta problem redan under sin tid i Cambridge och började nu utveckla ett lämplig verktyg, som med tiden kom att resultera i programspråket C++.

C++ utformades ursprungligen för att kunna erbjuda Simulas möjligheter att organisera program med hjälp av klasser, i kombination med C:s effektivitet och flexibilitet för systemprogrammering.

Inledningsvis kallades språket *C with Classes*. Det var först 1983 som språket gavs namnet C++. Språket har hela tiden utvidgats med nya tillägg. Det har även förekommit att saker tagits bort. I efterhand kan det tyckas att en genomtänkt design fått stryka på foten för experimenterande. Man kan också tycka att C++ har blivit större och mer komplicerat än det skulle behöva vara, om man ser till dess innehåll. Det finns t.ex., enbart för att nämna några saker, två olika bibliotek för in- och utmatning, två olika sätt att hantera strängar resp. vektorer och tre olika syntaxer för typomvandling.

I den första versionen av C with Classes (1980) ingick, förutom C-delen, klasser, arv (men inte virtuella funktioner), åtkomstspecifikation med **public** och **private**, konstruktorer och destruktorer, vänklasser (**friend**), typkontroll och typomvandling av funktionsargument. I 1981 års version tillkom **inline**-funktioner, förvalda värden för funktionsparametrar, samt möjlighet att överlagra tilldelningsoperatorn. Andra betydande konstruktioner som kan nämnas är strömbiblioteket för in- och utmatning som kom 1985, mallar (**template**) och undantag (**exception**) som kom 1990, dynamiskt typidentifiering (Runtime Type Identification, RTTI) och namnrymder (**namespace**) som kom 1993. Det nya standardbiblioteket enligt den 1997 antagna standarden är också ett stort tillägg.

Den första användbara implementeringen av C++ kom 1983 och språket blev allmänt tillgängligt 1985. Kommersiella system för C++ på olika plattformar dök successivt upp från 1986 och framåt. Flera av de stora först under 1990-1992. GNU:s första version av C++ kom 1987.

Den första manualen för C++ kom 1984. Stroustrups första upplaga av boken The C++ Programming Language kom 1985 och utgjorde i praktiken referensmanual. The Annotated C++ Referens Manual (ARM) kom 1990 och blev en viktig milstolpe i definitionen av C++. Stroustrups andra upplaga av The C++ Programming Language kom 1991. År 1994 lades ett förslag till ANSI/ISO-standard fram, vilken antogs i slutet av 1997. Den tredje upplagan av The C++ Programming Language kom 1998.

Övergripande jämförelse mellan Java och C++

Java är ett rent objektorienterat språk. En liten avvikelse från detta är att objekt av de enkla datatyperna hanteras på ett sätt som är det gängse i procedurella språk. Sådana objekt tilldelas minne automatiskt när exekveringen går in i deklarationsblocket och minnet återlämnas automatiskt när exekveringen lämnar blocket. Klass- och fältobjekt hanteras däremot alltid via referenser och tilldelas alltid dynamiskt minne. Ett referensobjekts minne återvinns automatiskt av lumpsamlaren (*garbage collector*) när det inte längre finns någon variabel som refererar till objektet.

Ett javaprogram består av klasser och allt måste deklareras inuti klasser. En klass är alltid en textuellt sammanhållen enhet, dvs den kan inte delas upp i en specifikationsdel och en implementeringsdel.

Klassobjekt hanteras alltid via referenser. En variabel som representerar ett objekt av klasstyp är alltså egentligen en variabel som lagrar adressen till objektet. För att komma åt objektet, t.ex. för att operera på en datamedlem eller anropa en metod, avrefereras referensen med punktnotation. Om en referensvariabel är **null**, en tomreferens, är det ett allvarligt fel att försöka avreferera den (och i språket som sägs sakna pekare genereras undantaget NullPointerException!).

Metoder binds alltid dynamiskt då de anropas så vid metodanrop i Java gäller alltid polymorfi.

Javas standardklasser är alla, direkt eller indirekt, härledda från klassen Object och ingår följaktligen i en stor gemensam klasshierarki. Dessutom organiseras standardklasserna utifrån funktionalitet i en paketstruktur, som inte alls är densamma som arvshierarkin. Man kan skapa egna paketstrukturer för egna samlingar av klasser.

I Java finns inga fria funktioner. Varje funktion, eller metod som de kallas i Java, är medlem i en klass. Metoder som är **static**-deklarerade kallas klassmetoder och kan användas utan att vi behöver skapa klassobjekt. Vi anropar i stället metoden med klassnamnet som prefix. Det är så nära fria funktioner vi kommer i Java.

Java har ett stort standardbibliotek med klasser för många slags tillämpningar. Detta är objektorienterat och alla klasser har en gemensam rot i klassen `Object`.

C++ är ett hybridspråk med både procedurella och objektorienterade egenskaper. I grunden är minnesmodellen densamma som gäller för Javas enkla datatyper, även avseende klassobjekt. För att hantera klassobjekt på ett sätt som överensstämmer med det i Java, måste objekten uttryckligen hanteras via pekare. Minnesåterlämningen måste programmeras, eftersom det inte finns någon lumpsamlare.

Det finns något som heter referenser i C++. Sådana motsvaras dock snarast av att man i Java skulle binda en referensvariabel till ett visst objekt när referensvariabeln skapas, för att sedan inte kunna binda om den till ett annat objekt. Referenser i C++ används främst som funktionsparametrar för att åstadkomma referensanrop, men även som t.ex. returvärden från funktioner och som medlemmar i klasser för att referera till andra klassobjekt.

Det finns i C++ ingen fördefinierad klass motsvarande `Object` i Java. En klass i C++ kan vara helt fristående från andra klasser. Om, och hur, man vill skapa klasshierarkier bestämmer man själv.

En klass i C++ kan definieras på olika sätt. En möjlighet är att skriva medlemsfunktionernas definitioner i klassdefinitionen, som i Java. En annan möjlighet är att enbart ange medlemsfunktionernas deklarationer i klassdefinitionen och skriva deras definitioner separat, på annan fil.

I C++ finns fria funktioner. En sådan funktion är alltså är inte medlem i någon klass, utan definieras på filnivå.

I C++ finns namnrymder, som är en enkel form av modulkonstruktion och som i kan jämföras med Javas paketkonstruktion.

C++ har ett stort standardbibliotek. Det kan inte sägas vara objektorienterad, även om det finns en hel del klasser och även en del klasshierarkier. Biblioteket består också av fria funktioner, de flesta från standardbiblioteket för C. De nyare och mer omfattande delarna av standardbiblioteket bygger i grunden på klass- och funktionsmallar (**template**) och mycket av funktionaliteten kring klassmallarna bygger på iteratorbegreppet.

Programstruktur, översättning och exekvering av program

Ett javaprogram består av klasser. I princip skrivs varje klass på en egen fil. Ett filnamn överensstämmer normalt med namnet på klassen som finns på filen med tillägg av suffixet `.java`. Ibland har man flera klasser på samma fil men då är alla utom en vanligtvis hjälpklasser till den (enda) publika klassen i filen.

Javakompilatorn översätter källkoden till en form lämpad att utföras av javatolken. Den koden skrivs på en fil med samma namn som källkodsfilen men med suffixet `.class`. Javatolken utför koden på `class`-filerna då den kör ett program. Java får därmed anses vara ett interpreterat språk, även om det alltså finns en kompilator som översätter från källkod till interpreterbar kod. I vissa tolkar görs översättning från den interpreterbara koden till maskinkod (”just-in-time”-kompilering), t.ex. av programdelar som visar sig utföras ofta, s.k. ”hot-spots”.

En Java-klass tillhör alltid ett paket, om inte annat ett anonymt paket. T.ex. tillhör alla standardklasser ett visst paket. Paketstrukturen är hierarkisk och ger en strukturerad organisation av alla klasser.

När man i Java vill använda en klass ur standardbiblioteket anges detta vanligtvis med en **import**-sats. En **import**-sats kan antingen kan avse alla klasser i ett helt paket eller en enskild klass. Alternativet är att skriva fullständiga namn i koden, t.ex. `java.util.StreamTokenizer`. Detta gäller även då man vill använda andra klasser än sådana som tillhör standardbiblioteket.

Ett C++-program kan bestå av klasser och/eller fria funktioner. Åtminstone en fri funktion finns alltid i ett C++-program. Den heter `main` och motsvarar metoden `main` i Java.

Ett C++-program kan delas upp på flera filer. Vad filerna kan innehålla och vad de ska heta är inte lika strikt reglerat som i Java. En klass delas typiskt upp på två filer. Klassdefinitionen med enbart deklaration av medlemsfunktionerna (metoderna) skrivs på en *inkluderingsfil*. Definitionerna för medlemsfunktionerna skrivs på en tillhörande implementeringsfil. Detta är en uppdelning av klasser som ej är möjlig i Java. Filerna ges vanligtvis samma namn som klassen, där typiskt inkluderingsfilen har suffixet `.h` (h som i *header*) och implementeringsfilen t.ex. suffixet `.cc` (Unix) eller `.cpp` (PC).

En annan vanlig situation är att man har en datatyp av något slag och en tillhörande mängd fria funktioner för att operera på objekt av datatypen ifråga. I sådana fall placerad datatypdefinitionen och funktionsdeklarationerna på en inkluderingsfil och funktionsdefinitionerna på en tillhörande implementeringsfil. Det här sättet att göra filuppdelning brukar kallas ”file as module”. Inkluderingsfilen utgör det publika gränssnittet för ”modulen” och implementeringsfilen utgör den dolda implementeringen.

I C++ finns namnrymder (**namespace**) som kan användas för att modularisera program. En namnrymd utgör ett namngivet deklarationsblock som kapslar de deklarationer som finns i blocket. Namnrymder motsvarar på sätt och vis paketen i Java men har inte den koppling till filstruktur som javapaketen har.

C++-program översätts av en kompilator till objektkod (objektkod skrivs på filer med suffix .o). Objektkodsfilerna som utgör ett program sätts sedan ihop med exekveringssystemet (runtime) av länkaren till ett körbart program. Om inget annat anges erhålls det körbara programmet på filen `a.out`.

En preprocessor behandlar alltid källkoden innan den kompileras. Den behandlar speciella preprocessordirektiv och utdata från preprocessorn är ren källkod i C++. En typisk uppgift för preprocessorn är att ersätta inkluderingsdirektiv med källkoden i angivna inkluderingsfiler. Detta motsvarar i princip **import**-satserna i Java men dessa har ju i Java enbart deklarativ effekt.

Följande är ett exempel på ett enkelt C++-program.

```
#include <iostream>      // ger tillgång till strömbiblioteket för in- och utmatning
#include <string>        // ger tillgång till strängklassen string
using namespace std;     // öppnar namnrymden för alla standardbiblioteksnamn

int main()
{
  string  message(”Hello World!”);     // en variabel av strängtyp deklareras och initieras
  cout << message <<endl;              // variabeln message skrivs ut
  return 0;                            // main returnerar ett diagnostiskt heltalsvärde
}
```

Om man inte använder **using**-direktivet måste standardnamn föregås av prefixet `std::`.

Syntax

Ytligt sett liknar Java och C++ varandra mycket. Tittar man närmare upptäcker man dock påtagliga skillnader. Det finns t.ex. 62 reserverade ord i C++ jämför med endast 50 i Java. Dessutom finns ca 15 av Javas reserverade ord inte i C++ eller saknar direkt motsvarighet. Det finns också en del smärre skillnader, t.ex. heter booletypen **boolean** i Java, **bool** i C++.

Den form av dokumentationskommentar, `/**…*/`, som finns i Java, saknas i C++.

Datatyper

De datatyper som finns i ett språk och hur starkt typat språket är, är avgörande för ett språks egenskaper.

I Java skiljer man i grunden på två slags datatyper, *enkla typer* och *referenstyper*. De enkla typerna utgörs av **boolean** respektive aritmetisk typer. De aritmetiska typerna delas i sin tur upp i heltalstyper, där även **char** ingår, samt flyttalstyper. Till referenstyperna hör klasser, uppräkningstyper, gränssnitt och fält. Gemensamt för referenstyperna är att sådana objekt alltid tilldelas minne dynamiskt och att de aldrig innehålls av variabler, utan hanteras via variabler som innehåller referenser till objekten. Referenstypernas hantering liknar det som man gör med pekare i andra språk.

I C++ är bilden mer komplicerad. De s.k. *inbyggda typerna* i C++ är i princip desamma som de enkla datatyper som i Java, men med fler varianter och inte lika väldefinierade värdemässigt. *Användardefinierade typer* kallas sådana som konstrueras från grunden eller utifrån de inbyggda typerna. Hit räknas uppräkningar (**enum**), pekare, referenser, fält, ”strukturer” (**struct**, i princip detsamma som klass), klasser (**class**) och unioner (**union**). Fält hanteras lite speciellt och står i nära relation till pekare.

Alla slags datatyper i C++ kan hanteras med automatisk tilldelning och återlämning av minne för objekten, på samma sätt som gäller för de enkla datatyperna i Java. Variablerna innehåller i detta fall objekten och inte referenser eller liknande. Fält är dock inte sådana s.k. första klassens objekt, utan en fältvariabel innehåller enbart adressen till det första elementet i fältobjektet.

Alla slags datatyper i C++ kan tilldelas minne dynamiskt. Objekten hanteras då via pekare på ett sätt som i princip överensstämmer med referenstyperna i Java. En viktig skillnad är dock att det inte finns någon lumpsamlare (garbage collector) i C++. Detta innebär att allt dynamiskt minne måste uttryckligen återlämnas för att inte s.k. minnesläckor ska uppstå, en vanlig källa till fel i C/C++-program.

Ett tredje sätt att hantera objekt i C++ är via referenser. En referens i C++ innehåller adressen till ett objekt men den är en *konstant* referens. En sådan referens måste bindas till ett objekt då den skapas och den kan sedan inte bindas om till något annat objekt. Objektet som refereras till på detta sätt skapas nästan uteslutande för någon annan variabel och en referens i C++ är därför snarast att betrakta som en synonym, ett alias. Den vanligaste användningen av referens är som funktionsparametertyp eftersom detta är sättet i C++ att erhålla referensanrop.

Appendix innehåller översikter av inbyggda och användardefinierade detatyper.

Enkla datatyper

Enkla datatyper i Java

Java har 8 enkla datatyper: **boolean**, **char**, **byte**, **short**, **int**, **long**, **float** och **double**. Av dessa är **byte**, **short**, **int**, **long** och även **char** heltalstyper, **float** och **double är** flyttalstyper. Till skillnad från många andra språk är de numeriska typernas minnesutnyttjande och värdeintervall väldefinierade i Java. Alla heltalstyper utom **char** kan anta negativa värden Om man anger **final** i en deklaration anger detta ett konstant objekt (**const** finns bland de reserverade orden i Java men har för närvarande ingen funktion).

Enkla datatyper i C++

C++ har betydligt fler enkla datatyper än Java och de är inte lika väldefinierade som i Java. Det finns en boolsk typ, ett flertal teckentyper, många heltalstyper, uppräkningstyp, tre flyttalstyper och en typ **void** motsvarande den i Java. Booletypen, teckentyperna, heltalstyperna och uppräkningstyper kallas gemensamt för *integral types*. Integral types och flyttalstypernas utgör *aritmetiska typer*. Samtliga dessa typer kallas för de *grundläggande datatyperna* (*fundamental types*).

Man kan definiera uppräkningstyper, **enum**.Detta är en *användardefinierad typ* men samtidigt en enkel typ. Uppräkningstyper har mycket gemensamt med **int** och används i princip som **int**. I Java är uppräkningstyper en slags klasstyp.

Det finns alltså en stor mängd enkla inbyggda datatyper i C++. Många är varianter av någon typ, t.ex. av heltalstypen **int**. I de flesta fall kan man använda **bool** för logiska värden, **char** för tecken, **int** för heltalsvärden och **double** för flyttalsvärden. Övriga typer är främst avsedda för optimering och för speciella behov.

Även om varje enkel datatyp är en distinkt typ, finns det många automatiska typomvandlingar, både värdebevarande och värdeförstörande, som gör typningen i princip svag. Det finns dessutom en hel del operationer på de inbyggda typerna som är tillåtna men där resultatet per definition ej ärodefinierat.

Booletyp

bool är en datatyp med värdena **false** och **true**, som används för resultaten för logiska uttryck. Per definition har **false** värdet 0 och **true** värdet 1. Omvänt kan heltalsvärdet 0 omvandlas till **false** och heltalsvärden skilda från 0 kan omvandlas till **true**. I aritmetiska och logisk uttryck omvandlas **bool**-värden till **int**, och logiska operationer utförs på de omvandlade värdena. Om resultatet omvandlas tillbaka till **bool**, blir värdet 0 omvandlat **false** och ett värde skilt från 0 omvandlat till **true**. Även pekarvärden kan omvandlas till **bool**. En nollpekare omvandlas till **false** och en icke-nollpekare omvandlas till **true**. Man ser rätt ofta missbruk av dessa typomvandlingar, i vissa kulturer upphöjda till accepterade idiom.

Teckentyper

char är en teckentyp som består av vanligtvis 8 bitar. **char**-värden kan omvandlas till **int**, vilket reser frågan om **char** har en teckenbit eller ej; tolkas de 256 värdena som 0 till 255 eller som -127 till 128? Dessvärre definieras detta inte av språket utan av implementationen, vilket kan leda till portabilitetsproblem. Storleken på **char** är minst 8 bitar och typen för **char** ska vara den mest lämpade för att hantera teckenvärden på den aktuella maskinen.

Utöver den enkla teckentypen **char** finns **unsigned char** och **signed char**, för vilka förekomsten av teckenbit är bestämd. Teckentyperna är s.k. *integral types*, vilket innebär att både heltalsoperationer och logiska operationer kan appliceras på teckenvärden, där det värde som då används är teckenkoden.

Teckenlitteraler skrivs inom apostrof, t.ex. ’A’. Det finns vissa speciella tecken vars namn skrivs med bakstreck, t.ex. ’\n’ för radslut och ’\0’ för strängslut i *C-strängar*, vilka representeras som teckenfält.

För större teckenmängder, som t.ex. 16-bitars Unicode, finns teckentypen **wchar_t** (det konstiga namnet är en kvarleva från C).

Heltalstyper

Den grundläggande heltalstypen är **int**. Den finns dock i tre former: vanlig **int**, **unsigned int** resp. **signed int**. De två senare används för att uttryckligen ange om ett **int**-värde ska ha en teckenbit eller ej. En vanlig **int** är alltid underförstått **signed**. Varianten **unsigned** är t.ex. lämplig om man ska använda ett lagringsutrymme som ett bitfält. Att försöka säkerställa att värden är positiva genom **unsigned**, kommer typiskt att saboteras av reglerna för automatisk typomvandling.

Dessutom finns **int** i tre storlekar: **short int**, **int** och **long int**. Enbart **short** kan användas som synonym för **short int**, och enbart **long** kan användas som synonym för **long int**. Det finns regler för hur många bitar dessa typer ska ha minimalt och maximalt och även för inbördes storleksförhållanden, som åtminstone säkerställer att t.ex. en **long int** inte kan vara mindre än en **short int**. Storleken för **short** ska vara minst 16 bitar och för **long** minst 32 bitar. Typen för **int** ska vara den mest lämpade typen för att hantera heltalsvärden på den aktuella maskinen.

Heltalslitteraler finns i fyra former: *decimal*, *oktal* (bas 8), *hexadecimal* (bas 16) samt *teckenlitteraler*. En oktal litteral inleds med 0 (noll). En hexadecimal litteral inleds med 0x, och bokstäverna `A` till `F` (eller `a` till `f`) används för att representera de decimala värdena 10 till 15. Teckenlitteraler skrivs inom apostrof, se avsnittet om teckentyper. Suffixet `U` kan användas för att uttryckligen ange en **unsigned** litteral, t.ex. `10U`, och suffixet `L` för att ange **long**, t.ex. `10L`.

Flytalstyper

Flyttalstyper finns i tre storlekar: **float** (enkel precision), **double** (dubbel precision) och **long double** (utvidgad precision). Den exakta meningen med enkel, dubbel och utvidgad precision definieras av implementeringen. Att välja rätt kräver stora kunskaper om flyttalsberäkning och har man inte det är **double** det bästa valet.

Flyttalslitteraler skrivs med decimalpunkt, t.ex. `1.23`, och med eller utan exponentdel, t.ex. `1.23e-2`. Sådana litteraler är underförstått av typen **double**. Om man vill ange att en flyttalslitteral ska vara av typen **float**, anges det med suffixet `f` eller `F`, t.ex. `1.23F`.

Typen **void**

Typen **void** är identisk med **void** i Java. I C++ kan **void** också användas som del i mer komplicerade typer, t.ex. för att låta en funktion returnera en pekare till ett objekt av godtycklig typ, **void***.

Symboliska konstanter

Konstanter för alla slags datatyper, inbyggda som användardefinierade, kan definieras med **const**., t.ex.:

```
const int     MAX_SIZE = 1000;
const double  EPSILON  = 0.1e-6;
```

Deklaration med **const** i C++ motsvarar deklaration med **final** i Java.

Deklarationer

Då det gäller deklarationer som rör klasser och deras medlemmar är reglerna i C++ och i Java i grunden lika. I C++ tillkommer ytterligare aspekter, genom att man kan deklarera fria funktioner och även variabler och konstanter på filnivå, samt språkregler som rör deklarations- och användningsordning.

Vissa deklarationer utgör *definitioner*, vilket innebär att de även definierar det som namnet refererar till. För en datatyp är det den fullständiga typbeskrivningen. För en variabel innebär det hur mycket minne som behövs för variabeln och att detta minne kommer att skapas. För en konstant är det värdet. För en funktion är det den fullständiga funktionsdeklarationen. Det måste finnas exakt en definition för varje namn i ett program.

Innan ett namn kan användas i ett C++-program måste det deklareras. Det innebär att dess typ måste anges, för att informera kompilatorn om vad namnet avser och därigenom hur namnet är tillåtet att använda. Några exempel på deklarationer, som i många fall även är definitioner:

```
char    c;          // en teckenvariabel (också definition av c)
string  s;          // en strängvariabel (också definition av s)
int     i = 1;      // en heltalsvariabel som initieras (uppenbarligen en definition)

extern int  i;      // en externdeklarerad heltalsvariabel (ej definition)

const double PI = 3.1415926535897932385;  // en konstant (definition)

enum Direction {UP, DOWN};                // typdefinition

struct Pair { int x, y; };                // typdefinition
struct Pair;                              // typdeklaration

int get_x(Pair p) { return p.x; }         // funktionsdefinition
int get_x(Pair);                          // funktionsdeklaration

typedef int* int_pointer;                 // typdefinition (inför synonym för int*)

namespace N { int x; }                    // definition av en namnrymd N
```

I C++ kan det ibland vara nödvändigt att deklarera t.ex. en datatyp (t.ex. en klass) för att kunna använda den, innan dess definition har kunnat göras. Någon motsvarighet finns ej i Java, utan sådana aspekter löses av kompilatorn; vill man använda en klass, gör man det helt enkelt.

Deklaration av medlemmar i klasser (och *structures*) behandlas inte i någon större utsträckning här, eftersom det i princip överensstämmer med vad som gäller i Java, även om det finns en hel del skillnader i detaljerna. Exempel ges senare. I följande avsnitt tar vi i stället upp några av möjligheterna att deklarera namn som är speciella för C++.

Deklarationer på filnivå

I C++ kan man definiera och deklarera namn på filnivå. En funktion kan definieras utan att vara medlem i en klass eller i en namnrymd. Den möjligheten är inte så speciell, utan förekommer i många språk. Även variabler och konstanter kan definieras på filnivå, vilket är mindre vanligt. Många språk kräver att dataobjekt deklareras i någon deklarativ del av någon programenhet, t.ex. i ett underprogram eller i någon form av modul.

Definitioner på filnivå är i princip globala i programmet. Man kan dock begränsa räckvidden till definitionsfilen genom att placera definitioner vars räckvidd ska begränsas till definitionsfilen i en s.k. *anonym namnrymd*:

```
…
namespace {
  int  i;
  void fun(int i) { … }
}
```

Övrigt filinnehåll…

Variabeln `i` och funktionen `fun` kan genom detta endast användas i andra deklarationer och funktioner som finns i samma fil. Ett alternativ (det enda som fanns före den nya standarden) som har samma effekt, är att deklarera med **static**:

```
static int  i;
static void fun(int i) { … }
```

Deklarerar man *inte* med **static** kan man i andra filer som tillhör programmet göra **extern**-deklarationer för att få tillgång till namn som definierats på filnivå.

```
extern int i;
extern void fun(int);
```

För funktioner är det egentligen inte nödvändigt, det går lika bra att göra en deklaration utan **extern**. Externdeklaration informerar kompilatorn om vad namnen står för och att de är definierade på annan fil.

Lokala deklarationer i funktioner

Precis som i Java kan funktioner ha lokala variabler. Deras räckvidd är funktionskroppen och deras livstid normalt densamma som funktionsaktiveringen, vilket är en effekt av att en lokal variabel tilldelas minne när funktionen aktiveras och att detta minne återlämnas när funktionen returnerar.

I C++ kan man **static**-deklarera en lokal variabel i en funktion. Det får som effekt att dess livstid blir densamma som programmets. Variabeln tilldelas minne och initieras när programexekveringen startar och den bibehåller sedan minnet till dess programmet avslutas. Variabelns värde kommer genom detta att bevaras mellan funktionsaktiveringar. Exempel:

```
void print_page_number(ostream& os) {
  static int page_number = 0;
  os << ++Page_number;
}
```

Den statiska variabeln `number` får sitt minne då programmet startas och initieras i samband med detta, och endast då. Variabeln kommer inte att nollställas vid varje anrop, utan ha det värde som den hade sedan tidigare.

Initiering

I princip initieras alla variabler som definieras utanför funktioner, dvs globala variabler, namnrymdsvariabler och statiska klassvariabler, innan huvudfunktionen `main`() anropas. Om ingen uttrycklig initierare deklarerats, initieras variabler till något standardvärde för typen ifråga. För inbyggda typer och uppräkningstyper är detta standardvärde 0 eller motsvarande.

Lokala variabler i funktioner och datamedlemmar i klasser initieras ej standardmässigt. I en klass kan medlemmarna uttryckligen initieras av klassens konstruktor. En medlem i en klass som inte uttryckligen initieras av klassens konstruktor men som är ett klassobjekt kan initieras av en egen standardkonstruktor.

Mer om deklarationer

Vi har ovan behandlat två slags **static**-deklarationer specifika för C++. I C++ kan **static** även användas för en medlem i en **class** (eller **struct**) och innebär då, precis som i Java, att medlemmen blir en klassmedlem, dvs inte associerad med något objekt.

I C++ finns möjlighet att ange **register** i en variabeldeklaration. Det anger att variabeln, om möjligt, ska placeras i ett register för att kunna opereras på effektivt. Detta bör man undvika och i stället låta kompilatorn sköta optimeringen, vilket den ofta gör bättre (i värsta fall kan man erhålla sämre prestanda). Java har ingen motsvarighet till **register**.

Deklaration med **volatile** innebär att en variabels värde kan ändras på sätt som ligger utanför programmets kontroll. Detta innebär i praktiken att varje gång en variabel ska användas, måste dess värde hämtas från eller sparas i globalt minne. Det kan inte antas att en variabels värde som ligger i ett register eller i cache-minne överensstämmer med värdet det i globala minnet. Det finns även i Java möjlighet att deklarera variabler som **volatile**.

En variabel som är medlem i en klass kan deklareras som **mutable**. Det innebär att en konstant medlemsfunktion, som normalt inte får ändra på ett klassobjekts status, tillåts ändra på en sådan variabels värde. Sådana variabler används vanligtvis för internt administration och deras värden syns normalt inte utåt.

Vanliga variabler som deklareras i block kallas *automatiska*, vilket syftar på den automatiska minneshanteringen för sådana variabler. Det finns ett reserverat ord **auto** som kan ges i sådana variabeldeklarationer men det gör man av tradition aldrig.

Någon motsvarighet till Javas **synchronized** eller **transient** finns ej.

Användardefinierade datatyper

Användardefinierade datatyper kan konstrueras genom *uppräkning* eller med de inbyggda datatyperna. De datatyper som kan konstrueras på det senare sättet omfattar *pekare* (`*`), *referenstyper* (`&`), *fält* (`[]`, array), *structures* (**struct**; en posttyp) och *klasser* (**class**). Observera att **struct** och **class** i princip är helt identiska frånsett en ringa detalj, nämligen det förvalda åtkomstskyddet för medlemmarna. Om inget åtkomstskydd specificeras gäller i **class** underförstått **private**, i **struct** underförstått **public**.

Uppräkningstyper, pekare och referenser är till sin struktur enkla, i den meningen att deras värden är skalära värden. Uppräkningsvärden är att betrakta som symboliska heltalskonstanter och användningen av uppräkningsvariabler liknar användningen av heltalsvariabler.

Pekare och referenser är adresser. Det som skiljer pekare och referenser är främst två saker. En referens måste alltid bindas till en adress då den skapas och den kan därefter inte ändras, den kan ses som en konstant pekare. När man opererar på en referens sker alltid underförstått avreferering, dvs operationen avser alltid det refererade objektet. Det sker aldrig några operationer på referenserna. Pekare däremot (som ej är deklarerade som konstanta) kan bindas om dynamiskt. När man opererar på en pekare väljer man att antingen operera på pekarvärdet eller på det objekt pekaren pekar på, beroende på om man avrefererar pekaren med någon av operatorerna `*` eller `->`, eller inte.

Övriga användardefinierade datatyper är sammansatta av ett eller (vanligtvis) flera element, som kan vara av olika datatyp. Fält består av element av samma datatyp, structures och klasser kan bestå av element av olika typ.

Uppräkningstyper - **enum**

En uppräkningstyp definieras genom uppräkning av de värden som ska ingå i typen, t.ex.

```
enum Direction {UP, DOWN, LEFT, RIGHT};
```

Detta definierar en distinkt datatyp `Direction`, som omfattar de värden som anges mellan parenteserna. Väl definierad används en uppräkningstyp i hög grad som en heltalstyp.

Uppräkningslitteralerna ska vara identifierare och de kan ej överlagras. Uppräkningsvärden representeras av heltal, normalt 0, 1, 2, osv. Ett uppräkningsvärde kan typomvandlas till **int** automatiskt, men för att typomvandla från **int** till uppräkningsvärde krävs en uttrycklig typomvandling:

```
{
  Direction  dir = UP;
  int  i = dir;          // automatisk typomvandling från Direction till int
  dir = Direction(3);    // uttrycklig typomvandling: 3 till LEFT
}
```

För uppräkningstyper kan man definiera egna operationer genom operatoröverlagring, t.ex. ++ för stegning eller << för utskrift.

```
ostream& operator<<(ostream& os, Direction d) {
  switch (d) {
    case UP:    os << "UP";
    case DOWN:  os << "DOWN";
    case LEFT:  os << "LEFT";
    case RIGHT: os << "RIGHT";
  }
  return os;
}
```

Fält - `[]`

Fält i C++ har stora likheter med fält i Java, men saknar `length` och kan deklareras statiskt. Det finns också lite syntaktiska skillnader, t.ex. kan inte `[]` anges direkt efter datatypen, som i Java, utan måste alltid ges efter variabelnamnet.

Fält som deklareras statiskt får sitt minne automatiskt tilldelat då exekveringen går in i deklarationsblocket. Följande definierar en variabel `vector` med minne för 100 heltal automatiskt tilldelat vid inträdet i deklarationsblocket och automatiskt återlämnat vid utträdet ur deklarationsblocket.

```
int  vector[100];
```

Fält som skapas dynamiskt hanteras via pekarvariabler. I exemplet nedan anger **int**`*` att variabeln `ip` är en ”pekare till **int**”:

```
int*  ip = new int[100];
```

Det framgår alltså inte direkt att `ip` är avsedd att peka på ett fält, det skulle kunna vara vilket **int**-objekt som helst.

En fältvariabel som `vector` ovan är inte ett första klassens objekt, utan utgör en konstant adress till det första elementet i objektet. I princip är `vector` likvärdig med `ip`, men kan inte bindas om till ett annat objekt, vilket `ip` kan. Pekarvariabeln `ip` kan för övrigt bindas till `vector` på följande sätt (en fälttyp som t.ex. T`[]` omvandlas i olika lägen till en motsvarande pekartyp T`*`).

```
ip = vector;
```

Structures - **struct**

struct är snarast att betrakta som en kvarleva från C, motiverad av bakåtkompatibilitetsskäl. Dock är den avsevärt modifierad jämfört med C:s **struct**, genom att i C++ att har givits samma egenskaper som **class**. De enda skillnaderna mellan **struct** och **class** är dels åtkomsten till medlemmarna i klassen i fråga, dels åtkomsten till medlemmar i en basklass, i samband med härledning. För **struct** gäller **public** om ingen åtkomstspecifikation uttryckligen anges, medan det är **private** för **class**. Klassens egenskaper beskrivs mer ingående nedan. Följande är ett exempel på en enkel **struct**:

```
struct Item {
  int    number;
  char   name[20];
  float  price;
  int    in_stock;
};
```

Unionstyper - **union**

En **union** är en **struct**, i vilken alla datamedlemmar placerats på samma adress. Det innebär att ett unionsobjekt endast upptar så mycket minne som den största datamedlemmen kräver, och att enbart

värdet för en datamedlem i taget kan lagras. Ett motiv för att använda unioner skulle kunna vara att spara minne men det är nog vanligare att använda unioner för att konstruera variantposter av det slag som finns andra språk (*variant record* i Pascal, *discriminated record* i Ada) eller för andra speciella syften.

Vilket värde som för tillfället lagras i ett unionsobjekt finns det inget sätt att kontrollera, som det t.ex. finns i Algol68. Detta måste programmeraren se till att det hålls reda på, vilket görs ofta med hjälp av en uppräkningstyp. En vanlig kombination av datatyper i samband med unionstyper är följande exempel på.

```
enum Kind {CHAR, INT, DOUBLE};

union Value {
  char    c;
  int     i;
  double  d;
};

struct Entry {
  string  id;
  Kind    type;
  Value   v;
};
```

Posttypen `Entry` har en fast del, `id`, och en variantdel, `v`, vars aktuella värde hålls reda på med hjälp av ett objekt av uppräkningen `type`. Följande visar en funktion som skriver ut innehållet i ett `Entry`.

```
void print(Entry e) {
  cout << e.id << ": ";
  switch (e.type) {
    case CHAR:   cout << e.v.c;
    case INT:    cout << e.v.i;
    case DOUBLE: cout << e.v.d;
  }
}
```

Unionsobjektet medförde att vi måste skriva t.ex. `e.v.c` för att komma åt teckenvarianten, fast vi egentligen enbart vill skriva `e.c`. Att `c`, liksom `i` och `d`, tillhör variantdelen ska ju inte behöva framgå på detta uttryckliga sätt. Genom att använda en *anonym union* som deklareras på plats kan detta undvikas.

```
struct Entry {
  string  id;
  Kind    type;
  union {
    char    c;
    int     i;
    double  d;
  };
};
```

Den objektorienterade lösningen på detta är att deklarera en klass `Entry` som enbart innehåller de gemensamma delarna, i detta fall strängen `id`. Sedan gör man tre subklasser, en för respektive variant.

*Pekare - **

Pekar är snarlika Javas referenser. Ett pekarobjekt innehåller en adress till ett objekt, på samma sätt som en referens i Java. I Java är dock användningen av referenser begränsad i jämförelse med vad man kan göra med pekare i C++.

Pekartyper konstrueras med *, t.ex. int* för att definiera en pekare som kan peka på **int**-objekt. En pekare kan peka på en statiskt deklarerad variabel, genom att man med adressoperatorn & tar fram adressen till variabeln och tilldelar denna till pekaren. Detta tillåter inte Java.

För att avreferera en pekare finns operatorn *. Följande är inte speciellt meningsfullt men det visar hur det fungerar att ta adressen till en **int**-variabel `i`, lagra adressen i en **int**-pekare `ip`, och sedan operera på `ip` på olika sätt. En pekare som inte pekar på något objekt ska tilldelas värdet 0.

```
{
  int   i1 = 5;
  int*  ip = &i1;   // ip innehåller adressen till variabeln i
  *ip = 7;          // värdet på variabeln i1 ändras till 7
  int  i2 = *ip;    // variabeln i2 erhåller värdet av variabeln i1
  ip = 0;           // adressen i ip ändras till 0, tompekaren
}
```

Ett annat sätt att använda pekare är att låta dem peka på dynamiskt skapade objekt, vilket motsvarar vad man använder referenser till i Java. Objekt skapas dynamiskt med operatorn **new**, vilken motsvarar operatorn **new** i Java. I Java finns automatisk lumpsamling av referensobjekt.I C++ finns normalt inte någon lumpsamlare, utan det är programmerarens hela ansvar att återlämna minne för dynamiskt skapade objekt. Återlämning av dynamiskt minne görs med operatorn **delete**`[]` för fältobjekt, **delete** för alla andra slags objekt. Om ett fält innehåller objekt som *inte* har destruktorer som ska utföras, kan **delete** även användas för ett sådant fält.

Hanteras inte dynamiskt minne korrekt kan det antingen innebära att pekare refererar till minne som återlämnats (”dangling pointers”), eller att objekt tappas bort (”lost objects”). Dessa problem kan inte uppstå i Java, genom den restriktivare användningen av minne/adresser och lumpsamlaren.

Följande visar hur objekt skapas och återlämnas dynamiskt. Det visar också hur nära pekare och fält är associerade i C++. En pekare kan behandlas som om den vore ett fält och ett fält kan användas som om det vore en pekare.

```
{
  int*  ap = new int[100];  // skapar dynamiskt ett heltalsfält med 100 element
  for (int i = 0; i < 100; ++i) {
    ap[i] = 0;
  }
  …
  delete[] ap;  // återlämna det dynamiska minnet ([] ej nödvändigt här)
}
```

I C++ finns pekar/adressaritmetik, vilket innebär att aritmetiska operationer kan utföras på pekarvärden, t.ex. addition. Sådan aritmetik görs med hänsyn till pekartypen: för en pekare av typen **int*** innebär t.ex. addition med 1 att pekarens värde stegas fram motsvarande minnesstorleken för en **int**. **for**-satsen i exemplet ovan skulle alternativt kunna skrivas med pekararitmetik enligt följande:

```
for (int* ip = ap; ip < ap + 100; ++ip) {
  *ip = 0;
}
```

Slingvariabeln `ip` initieras med värdet av `ap`, som är adressen till det första elementet i fältet. Sedan kontrolleras att `ip` inte stegats förbi sista elementet i fältet, som är `ap` + `99`; `ap` + `100` är alltså adressen närmast efter sista elementet. Om så inte är fallet utförs satsdelen och sedan stegas `ip` med `++ip`. Pekararitmetik är numera ett föråldrat sätt att försöka källkodsoptimera, som kan omöjliggöra för moderna kompilatorer att applicera andra optimeringar, vilket kan resultera i sämre kod.

Referenser - &

En referens kan ses som en konstant pekare med automatisk avreferering. Den utgör i praktiken ett alternativt namn, ett alias, för det objekt den refererar till. Följande exempel är inte särdeles realistiskt men visar principerna för hur referenser fungerar.

```
int   i1 = 5;
int&  ir = i1;    // ir måste bindas i definitionen, adressen till i1 beräknas underförstått
ir = 7;           // variabeln i1 erhåller värdet 7
int  i2 = ir;     // variabeln i2 erhåller värdet av variabeln i1, dvs 7
```

En vanlig användning av referenser är att deklarera funktionsparametrar som referenser. Genom detta får man s.k. referensanrop, dvs att en ändring av parametern innebär att motsvarande arguments värde ändras, utan att behöva tillgripa pekare (som i C). Mer om detta i samband med funktioner.

En annan vanlig användning av referenser är att ha sådana som medlemmar i klassobjekt för att referera till andra objekt, t.ex. för att implementera en association.

En icke-konstant referens måsta alltid initieras med ett *lvalue* (ett variabelt objekt som man kan beräkna adressen för) av rätt typ; en **int&** måste initieras med adressen till en **int**-variabel. En konstant referens behöver dels inte initieras med ett *lvalue* och typomvandling får förekomma.

```
const double& cdr = 1;
```

Initieringen ovan kan tolkas enligt följande.

```
double  temp = double(1);
const double&  cdr = temp;
```

Det temporära objektet kommer att existera ända till dess räckvidden för referensen `cdr` lämnas.

Typdefinition – typedef

Deklaration med **typedef** är inte en typdefinition i vanlig mening, dvs deklaration en ny, distinkt datatyp. Det är snarare en deklaration av att alternativt namn, ett alias. **typedef** används typiskt för att deklarera ett enkelt namn för en komplicerad datatyp. Följande typdefinitioner deklarerar först ett enkelt namn, `List_Node_Pointer`, för datatypen **struct** `List_Node*`, och sedan datatypen `List`.

```
typedef struct List_Node*  List_Node_Pointer;
typedef List_Node_Pointer  List;
```

typedef kan användas även för att deklarera mer avancerade saker, som t.ex. en specifik instans av en klassmall och liknande, vilket kan öka läsbarheten avsevärt och reducera komplexiteten hos koden.

Typomvandling

I C++ finns tre syntaxer för typomvandling. En kommer från C och brukar kallas *cast*. Den innebär att den typ som man vill omvandla till skrivs inom parenteser före det som man vill typomvandla. dvs som i Java.

```
(int*)…
```

Parenteserna kring typen kan motiveras av att datatypnamn i C och C++ ofta utgörs av flera, ofta mellanrumsseparerade, delar. Detta kan utan parenteser leda till problem i en omgivning.

En andra syntax infördes tidigt i C++. Denna överensstämmer med hur typomvandling skrivs i många andra språk, dvs på funktionsform. Datatypen anges först och därefter skrivs det uttryck eller värde som ska typomvandlas inom parentes:

```
int(x + 0.5)
```

Den tredje varianten är införd relativt sent i C++. Stroustrup själv påstår att syntaxen är medvetet fult utformad för att motverka användning av uttrycklig typomvandling. Det finns fyra former av typ-omvandlingar att välja bland (de är för övrigt operatorer). Samtliga slutar på **_cast**, vilket också gjorts medvetet, så att man ska kunna söka på den strängen för att lätt hitta alla typomvandlingar.

static_cast används för typomvandling mellan relaterade typer, som t.ex. från en pekare till en annan pekare, från heltal till uppräkning, eller från flyttal till en heltal. Denna form av typomvandling är den som normalt används för vanlig typomvandling. I följande exempel är `malloc`() en C-funktion som tilldelar dynamiskt minne i enheter om ”byte” och returnerar **void*** (eller **char*** i gamla system):

```
double  d = 47.11;
int   i = static_cast<int>(d);                  // double till int

int*  p = static_cast<int*>(malloc(100));       // pekare till pekare
```

reinterpret_cast används för typomvandling mellan icke relaterade typer. Detta är en relativt ovanlig och rå typomvandling. Vanligtvis ger den ett värde med samma bitmönster som argumentet men med den önskade typen. Denna typomvandling används för implementationsberoende, farliga men ibland

nödvändiga typomvandlingar av heltalsvärden till pekare och tvärt om. I följande exempel antas `0xFF0` vara en minnesadress för en drivrutin.

```
IO_device*  io_dev1 = reinterpret_cast<IO_device*>(0xFF0);
```

Uppdelningen i **static_cast** och **reinterpret_cast** tillåter kompilatorn att göra viss kontroll av typomvandling med **static_cast** och göra det lättare för programmeraren att hitta de farligare typomvandlingarna som görs med **reinterpret_cast**.

dynamic_cast är en typomvandling med dynamisk typkontroll och den används för typomvandlingar av pekare och referenser som refererar till polymorfa, virtuella basklasser. Syftet med **dynamic_cast** är att hantera typomvandlingar vars korrektheten inte kan avgöras av kompilatorn och det motsvarar den typomvandling som brukar göras på referenser i Java. **dynamic_cast** kan användas för att typomvandla en basklass till en härledd klass (*downcast*) eller till en syskonklass (*crosscast*). Om `T` är en typ och `p` är en pekare kommer

```
dynamic_cast<T*>(p)
```

att returnera en pekare av typen `T*` *om* objektet som `p` pekar på är av typ `T` eller en subklass till `T`, annars returneras 0. Detta gör att **dynamic_cast** kan användas för att ta reda på om objektet som `p` pekar på är av typ `T`. Om `p` skulle vara 0, returneras 0. Ett vanligt sätt att använda **dynamic_cast** visas av följande:

```
T* t = dynamic_cast<T*>(p);
if (t != 0)
  t->f();  // anrop av medlemsfunktionen f() är endast tillåtet för subklassen T
else
  // inget objekt av typ T
```

Om `r` är en referens och objektet som refereras (det finns alltid ett objekt) är av typen `T` kommer

```
dynamic_cast<T&>(r)
```

att returnera en referens av typen `T&`. Om objektet inte är av typen `T` kommer undantaget `bad_cast` att genereras (att returnera 0 är varken möjligt eller önskvärt för referenser).

const_cast är en typomvandling för att ta bort konstanthet (*casting away const*) hos konstanta objekt.

Uttryck och satser

Mycket av det man kan skriva i form av uttryck och satser i Java är i många fall direkt gångbart även i C++. Utöver det finns det både mer syntax och mer regler att lära sig i C++.

Operatorer

Operatoruppsättningen och operatorsemantiken i C++ är mer omfattande än i Java. Detta beror främst på att C++ har

- uttrycklig adresshantering och speciella operatorer (`&`, `*`, `.*` och `->*`) för detta.
- pekare och en speciell operator (`->`) för att operera på objekt som refereras via pekare.
- en mer komplicerad hantering av dynamiskt minne med uttrycklig återlämning av dynamiskt minne och operatorer för detta (**delete**, **delete**[]).
- lågnivåhantering av minne som bygger på att man bestämmer minnesbehov för typer och uttryck i enheten byte, vilket görs med en operator (**sizeof**).
- *tre* olika syntaxer för uttrycklig typomvandling. Två av dessa får anses som föråldrade att använda i nyskrivna program och den nya omfattar fyra olika operatorer för olika slags typomvandlingar (**const_cast**, **dynamic_cast**, **reinterpret_cast**, **static_cast**).
- en räckviddsoperator (`::`) för att anropa medlemsfunktioner som är klassmedlemmar, för att ange fullständiga namn för medlemmar i namnrymder, samt för att kunna särskilja globala namn från lokala namn.
- en operator för att sätta samman en sekvens av uttryck, kommaoperatorn (`,`).

Java har två operatorer som inte finns i C++, logiskt högerskift (`>>>`) och den motsvarande sammansatta tilldelningen (`>>>=`). I Java är **throw** (för att generera undantag) en sats, medan det är en operator i C++.

I C++ tillkommer, som en extra dimension då det gäller operatorer, att man kan definiera egna operatoröverlagringar för egendefinierade datatyper.

Uppdelningen på prioritetsnivåer och associativiteten på respektive prioritetsnivå är i princip samma som för operatorerna i Java. En liten skillnad är att uttrycksoperatorn (`?:`) har lägre prioritet i C++ än tilldelningsoperatorerna, medan det är tvärt om i Java.

Appendix innehåller en operatoröversikt.

Satser

I C++ finns inte satsen **synchronized**, men för övrigt finns samma satser, inklusive blocksatsen. Några små men i vissa fall viktiga skillnader finns, mestadels till Javas fördel:

- i C++ finns ej någon motsvarighet till **try**-blockets **finally**-klausul.
- i C++ är **throw** en *operator*, inte en sats som i Java.
- i C++ kan *inte* en satsetikett (*label*) anges i **break**- eller **continue**-satser. Satsetiketter kan sättas in, men används endast tillsammans med **goto**.
- i C++ finns hoppsatsen **goto**. I den anger vi en satsetikett dit hoppet ska ske. Det finns regler för hur hopp får göras men den här typen av hopp ska endast användas i yttersta nödfall.
- I C++ är det tillåtet att definiera om ett variabelnamn i ett djupare nästlat satsblock.

För övrigt är det värt att påpeka, att i Java krävs att styruttryck i **if**-satser, **while**-satser, etc., är logiska uttryck (har typen **boolean**). I C++ finns inte detta krav, utan här accepteras även aritmetiska uttryck och vid behov sker automatiska typomvandlingar så att ett värde kan tolkas som sant eller falskt. Detta öppnar för stora möjligheter att skriva ful kod och att av misstag skriva uttryck som ger helt fel logik.

Appendix innehåller en översikt av satserna i C++ baserad på syntax.

Funktioner

I C++ förekommer funktioner, dels som medlemsfunktioner i klasser (kallas metoder i Java), dels som fria funktioner (finns inte i Java). En fri funktion hör inte till någon klass utan anropas direkt och de (icke-lokala) objekt som en sådan funktion kan operera på är antingen överförda via parametrar eller är globala objekt som är synliga för funktionen.

Frånsett förekomsten av fria funktioner i C++ finns det en hel del som överensstämmer med vad som gäller för metoder i Java. En del skillnader finns dock, t.ex. vad gäller parameteröverföring, vilket tas upp nedan.

Funktionsdeklaration och funktionsdefinition

En funktion i C++ kan delas upp i två delar (det gäller även medlemsfunktioner i klasser), som kallas *deklaration* resp. *definition*. En funktionsdeklaration består enbart av funktionshuvudet, dvs returtypen, funktionsnamnet och parameterlistan och avslutas med semikolon. En funktionsdefinition utgörs av den fullständiga funktionen.

```
int max(int x, int y);      // Deklaration
int max(int x, int y)     // Definition
{
  return (x > y ? x : y);
}
```

Parameteröverföring

Värdeanrop, dvs att ett arguments värde kopieras till motsvarande parameter och lagras i ett eget minnesutrymme associerat med funktionsaktiveringen, är den enda parameteröverföringsmoden i Java. Genom att alla klassobjekt hanteras via referenser innebär detta dock att klassobjekt i praktiken överförs på ett sätt som motsvarar referensanrop, dvs att adressen till argumentet överförs till parametern varigenom argumentet kan manipuleras via parametern. Om vi vill åstadkomma referensanrop för enkla datatyper i Java måste sådana värden placeras i en klass och referensen till ett sådan klassobjekt överföras.

Den grundläggande parameteröverföringsmoden i C++ är värdeanrop. Sedan kan man, som man måste göra i C, åstadkomma referensanrop genom att använda pekare och adress- och avrefereringsoperatorerna. Referensanrop görs dock enklare och elegantare i C++ med referenser. Fält är speciella i förhållande till övriga datatyper, genom att en fältvariabel inte innehåller själva fältet, utan endast adressen till det första elementet i fältet. Detta medför att överföring av fältvariabler i praktiken alltid fungerar som referensanrop och att det är fel att försöka göra uttryckligt referensöverföring.

Följande funktion visar fyra olika slags **int**-parametrar, en vanlig **int**, en referens till **int**, en pekare till **int**, samt ett **int**-fält.

```
void fun(int ic, int& ir, int* ip, int iarr[]) {
  ic = 10;
  ir = 20;
  *ip = 30;
  for (int i = 0; i < 100; ++i)
    iarr[0] = 0;
}

int main() {
  int  i = 1, j = 2, k = 3;
  int  a[100];

  fun(i, j, &k, a);
  …
}
```

I anropet överförs variabeln `i` genom värdeanrop till parametern `i`, dvs en kopia av `i`:s värde överförs till `ic` och sedan upphör all koppling mellan dessa objekt. Alla operationer på `ic` berör enbart `ic`.

Adressen till variabeln `j` överförs genom referensanrop till parametern `ir`. Alla operationer på `ir` blir i praktiken operationer direkt på `j`:s värde, genom den underförstådda avreferering som alltid gäller för referenser.

Adressen till variabeln `k` överförs genom värdeanrop till parametern `ip`, dvs adressen till `k` kopieras. Operationer på `ip` kan sedan antingen avse det lokala pekarvärdet i `ip` eller, genom avreferering av `ip` med `*`-operatorn, värdet i variabeln `k`.

Fältvariabeln `a` överförs genom värdekopiering till parametern `iarr`, vilket innebär att den adress som finns i `a` överförs till `iarr`. fältoperationer som utförs på `iarr` kommer därför att utföras direkt på värdena i fältet `a`.

För komplicerade objekt gör en funktionsparameter i form av en konstant referens det möjligt att undvika värdekopiering, med allt vad det kan innebära, men samtidigt förhindra att argumentet kan ändras.

```
void fun(const string& s)
{
  // s  kan användas men inte ändras
}
```

Detta är standardförfarandet för att överföra klassobjekt som inparametrar.

Förvalda parametervärden

I C++ kan man ge en parameter ett förvalt värde, vilket används om inget motsvarande argument ges i ett anrop. Något sådant finns ej i Java. Konstruktionen är något av en halvmessyr i C++, genom att man inte i anrop kan ange vilka formella parametrar som man uttryckligen vill binda genom uttryckliga argument, som man t.ex. kan göra i Ada. Se t.ex. avsnittet **Konstruktorer** nedan för exempel.

Inline-deklaration

Funktioner kan deklareras med **inline**. Det innebär att funktionsanrop inte görs på vanligt sätt, dvs med hopp till ett separat kodsegment för funktionen och användning av exekveringsstacken för att administrera anropet. I stället ersätts i princip funktionsanropet med de deklarationer och satser som finns i funktionskroppen. Syftet är att erhålla effektivare program, genom att undvika den administration som finns kring funktionensaktivering och återhopp. Något sådant finns ej i Java.

Klasser, arv, polymorfi

Java har en enkel modell för arv och en enhetlig semantik då det gäller att hantera och operera på klassobjekt. C++ har en mycket komplicerade arvsmodell med bl.a. multipelt arv. Vad som gäller i olika situationer, t.ex. med avseende på polymorfi, beror på hur man deklarerar medlemsfunktioner och hur man deklarerar/skapar klassobjekt. Någon motsvarighet till Javas anonyma klasser finns inte i C++, däremot kan man deklarera lokala klasser. Den här delen av C++ är mycket komplicerad och omfångsrik, både syntaktiskt och semantiskt. Det som följer nedan är endast en översikt på en grundläggande nivå.

Klasser

Följande klassdefinition i Java (som bör finnas på filen `Point.java`)

```
public class Point {
  private int  x;
  private int  y;

  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() { return x; }
  public int getY() { return y; }

  public void setX(int x) { this.x = x; }
  public void setY(int y) { this.x = x; }
}
```

kan i C++ definieras enligt följande (och lämpligtvis skrivas på en fil med namnet `Point.h`)

```
class Point {
public:
  Point(int x, int y) {
    this->x = x;
    this->y = y;
  }

  int getX() const { return x; }
  int getY() const { return y; }

  void setX(int x) { this->x = x; }
  void setY(int y) { this->x = x; }

private:
  int  x;
  int  y;
};
```

Den annorlunda placeringen av den privata delen är enbart föranledd av gängse konvention i modern C++-programmering. Observera att klassdefinitionen i C++ avslutas med ett semikolon, en liten men viktig detalj.

Deklarationen med **const** för medlemsfunktionerna `getX`() och `getY`() anger att dessa funktioner inte ändrar på objektet, dvs inte kan ändra på värdena för `x` och `y`. Det är viktigt att deklarera alla medlemsfunktioner som inte ändrar objektet som konstanta. Det är nämligen enbart konstanta medlemsfunktioner som får operera på konstanta objekt. Underlåter vi att konstantdeklarera sådana medlemsfunktioner, reducerar vi möjligheterna att operera på konstanta objekt. Konstantdeklaration av medlemsfunktioner delar upp operationerna i sådana som ändrar (kan ändra) på objekten (*mutators*) och sådana som garanterat inte ändrar på objekten (*selectors*). Det ger en direkt information till den som läser koden om olika medlemsfunktioners egenskaper i detta avseende. Genom att konstantdeklarera medlemsfunktioner som inte ska ändra på objekt kan kompilatorn kontrollera att detta verkligen uppfylls. Enbart andra konstanta funktioner får anropas från en konstant funktion, så det finns inga luckor i kontrollen.

this är som i Java en självreferens men i C++ är det en pekare av typen ”pekare till `Point`”, `Point*`, och den avrefereras därför med operatorn `->`. I en **const**-deklarerad medlemsfunktion är **this** i stället av

typen **const** `Point*`, dvs ”pekare till konstant `Point`”, och det är i praktiken så som regeln att **const**-medlemsfunktioner inte får ändra på objektet som anropat dem upprätthålls.

Konstruktorn skulle antagligen hellre skrivas enligt nedan, med s.k. *initierare* för att initiera datamedlemarna, vilket f.ö. är det enda sättet att initiera en datamedlem av klasstyp som kräver konstruktorargument.:

```
Point(int x, int y) : x(x), y(y) {}
```

Lite namnförbistring uppstår här genom att datamedlemmarna och parametrarna har givits samma namn, men det `x` som anges före parentesen avser datamedlemmen `x`, **this**`->x`, det `x` som står inom parentesen avser parametern `x`. Ett vanligt sätt att ha ”lika” men ändå olika namn är att inleda namnen på datamedlemmar med ett understrykningstecken, dvs `_x` och `_y` i detta fall.

Konstruktorer

Precis som i Java kan konstruktorer överlagras och det finns ett par speciella former av konstruktorer i C++. En konstruktor som kan *anropas* utan argument kallas *standardkonstruktor* (*default constructor*). Att den kan anropas utan argument betyder inte att den inte kan ha några parametrar, men om det finns parametrar måste i så fall samtliga ha givits förvalda värden. Konstruktorn för `Point` ovan kan göras om till en standardkonstruktor genom att deklarera den som:

```
Point(int x = 0, int y = 0) : x(x), y(y) {}
```

Man kan därmed deklarera `Point`-objekt utan att ange några initialvärden, men även ett eller båda:

```
Point  p1;                  // x och y initieras till sitt standardvärde, 0
Point  p2(1, 4);
Point  p3(3);               // x initieras till 3, y får anta standardvärdet 0
```

En annan form av konstruktor är *kopieringskonstruktorn*. Den används när ett klassobjekt ska användas för att initiera ett nytt klassobjekt av samma typ. Det används i uttryckliga deklarationer, som t.ex.

```
Point  p1;
…
Point  p2(p1);
```

Kopieringskonstruktorn används även vid parameteröverföring, då värdekopiering görs, och då objekt returneras från funktioner. Exempel:

```
Point fun(Point p)      // p kommer att initieras av kopieringskonstruktorn
{
  Point tmp = p;        // tmp kommer att initieras av kopieringskonstruktorn (!)
  …
  return tmp;           // kopieringskonstruktorn kommer troligtvis att användas
}
```

Då funktionen returnerar kommer, om ingen optimering lyckas eliminera detta, ett temporärt objekt att skapas och det initieras då med `tmp` av kopieringskonstruktorn.

Konstruktorer som har *en* parameter kan användas som *typomvandlande konstruktorer* och användas för automatisk typomvandling, om kompilatorn ser sådana möjligheter. Man kan eliminera sådan automatisk typomvandling genom att deklarera konstruktorer med en parameter som **explicit**.

Medlemsinitiering kan inte, som i Java, göras i samband med deklarationen av en medlemsvariabel, utan enbart i konstruktorerna. Det finns ett undantag som gäller statiska konstantmedlemmar av *integral*-typ eller uppräkningstyp, vilka kan initieras direkt i deklarationen. Någon motsvarighet till klassinitierare finns inte heller i C++.

Destruktorer

En destruktor i C++ motsvarar metoden `finalize`() i Java. Syftet är också i princip detsamma, nämligen att städa upp efter ett objekt då det försvinner. I Java är, p.g.a. skräpsamlarn, behovet av att använda `finalize`() betydligt mindre än behovet att använda destruktorer i C++. Varje klass i C++ som använder dynamiskt minne (klassen innehåller pekare) är normalt också en klass som ska ha en destruktor, för att säkerställa återlämnandet av objektens dynamiska minne.

En klass kan endast ha en destruktor (en destruktor kan inte överlagras). En destruktor kan inte ha någon returtyp, inte ens **void**, och inte heller några parametrar. Syntaxen för att deklarera en destruktor än ett ”tilde”-tecken (~) och klassens namn. En fragmentarisk klassdefinition med en destruktor:

```
class Array {
public:
  Array(int size = 100) : size_(size), element_(new double[size]) {}
  ~Array() { delete[] element; }

  …
private:
  int      size_;
  double*  element_;   // implementeras med ett dynamisk minnestilldelat double-fält
};
```

Exempel:

```
{
  Array  v(200);  // en Array med 200 element deklareras och initieras
  …
}   // räckvidden för v lämnas - v:s minne försvinner - destruktorn körs dessförinnan
```

Destruktorer körs också då man med **delete** återlämnar dynamiskt minnestilldelade objekt.

```
Array*  vp = new Array(50);
…
delete vp;   // destruktorn körs innan minnet återlämnas
```

En virtuell basklass ska alltid ha en virtuell destruktor, även om den inget gör. Annars kommer inte subklassers destruktorer att köras om sådana objekt refereras via pekare eller referens av basklasstyp.

Åtkomstspecifikation

I Java finns fyra åtkomstnivåer, **public**, **protected**, **private** och, om inget annat uttryckligen anges, *paketåtkomst* eller ”*friendly access*”. Åtkomstnivån anges för varje medlem, i dess deklaration.

I C++ finns tre åtkomstnivåer för medlemmar, **public**, **protected** och **private**. För en klass kan inget sådant skydd anges. Åtkomstskyddet för medlemmar anges genom att man delar in klassdefinitionen i avdelningar, se exemplet ovan, och varje medlem placeras i lämplig avdelning. Om inget anges är skyddet underförstått **private**. Åtkomstskyddet **public** innebär att vem som helst kan komma åt en sådan medlem, **private** att endast klassens egna medlemsfunktioner kan komma åt medlemmen och **protected** att enbart klassens egna samt eventuella subklassers medlemsfunktioner kan komma åt medlemmen.

Åtkomsskyddet kan åsidosättas genom vändeklarationer, **friend**. En vändeklaration innebär att en annan klass (alla dess medlemsfunktioner), eller en viss angiven medlemsfunktion i en annan klass, eller en fri funktion, ges direkt åtkomst till samtliga medlemmar i klassen ifråga, oavsett åtkomstskydd. Detta motsvarar *friendly access* i Java (det är f.ö. från friend i C++ sombegreppet *friendly* i Java kommer).

Hantering av klassobjekt

I Java deklarerar vi referensvariabler för att hantera klassobjekt och tilldelar alltid klassobjekt minne dynamiskt med operatorn **new**. Efter att sista referensen till ett objekt försvunnit kan lumpsamlaren återta minnet (*när* vet man dock inte och man kan inte tvinga fram det).

I C++ kan man deklarera klassobjekt statisk, dvs som variabler som är av klasstypen ifråga. En sådan variabel är alltså inte en referens eller pekare, utan innehåller objektet. Sådana variabler får sitt minne automatiskt tilldelat vid inträdet i deklarationsblocket och automatiskt återlämnat vid utträdet ut blocket.

```
{
  Point  p(1, 4);
  cout << ”x = ” << p.getX() << endl;
  …
}  // minnet för objektet p återvinns nu automatiskt
```

Den i C++ närmaste motsvarigheten till Java får vi i detta exempel genom att deklarera en variabel som är av typen pekare till `Point` och skapa objekt med operatorn **new**, se nedan. Ett sådant objekt måste, då det inte längre ska användas, återlämnas med operatorn **delete**, annars uppstår en s.k. minnesläcka.

```
{
  Point*  p = new Point(1, 4);
  cout << "x = " << p->getX() << endl;
  …
  delete p;     // minnet för det objekt p pekar på återvinns nu
}
```

Man kan i Java tilldela en referensvariabel värdet **null** då man inte längre har användning för objektet som den refererar till, för att därigenom möjliggöra för lumpsamlaren att återvinna minnet för objektet innan referensvariabeln försvinner (under förutsättning att ingen annan referens finns till objektet). Det är dock stor skillnad mellan ett sådant förfarande och att använda **delete** i C++.

Att Java saknar pekare i C++ mening eliminerar några klassiska problem, som på engelska brukar kallas ”dangling pointers” resp. ”lost objects”. *Dangling pointer* avser att man kan ha en pekare som pekar på ett objektet som egentligen inte längre existerar på grund av att dess minne är återlämnat. Det finns flera sätt att åstadkomma detta i C++. Ett sätt är med adressoperatorn `&`. Om vi i en funktion tar adressen till en lokal variabel (ett stackobjekt), sparar adressen i en icke-lokal pekare och returnerar från funktionen, har vi en illegal adress i pekaren. Nästa funktionsanrop kommer att förstöra objektet genom att objektets minnesutrymme på stacken då kan komma att skrivas över med något annat. Ett annat sätt att åstadkomma ”dangling pointer” är genom operatorn **delete**. Det finns genom denna möjlighet att efter **delete** på en pekare operera på det objekt som återlämnats, både via samma pekare eller via en annan pekare.

Lost objekt avser att man inte återlämnat minnet för ett objekt och alla pekare till objektet är borta. Objektet är förlorat och ligger kvar som skräp i ”heapen”, eftersom ingen lumpsamlare normalt finns.

Avsaknad av adressoperator och operationer för att uttryckligen återlämna minne tillsammans med lumpsamling eliminerar problemen med *dangling pointers* och *lost objects* i Java. Det kan dock betyda att man i vissa fall inte har den kontroll över minneshanteringen som man skulle behöva ha och även att man inte kan göra vissa saker som skulle ibland kan vara praktiskt eller nödvändigt. Logiska fel, som t.ex. att avreferera en **null**-referens, kvarstår naturligtvis i Java och leder till att ett undantag genereras, som av någon anledning heter `NullPointerException` (det är ju en *referens*).

Separatdefinierade medlemsfunktioner

I Java kan man inte dela upp en klassdeklaration i delar, t.ex. en specifikationsdel och en implementeringsdel. I C++ kan man välja att enbart *deklarera* medlemsfunktionerna i klassdefinitionen och *definiera* dem separat, vanligtvis på en annan fil. Klassen `Point`, se ovan, kan alternativt skrivas enligt följande. På en *inkluderingsfil* med namnet `Point.h` skrivs klassdefinitionen med enbart deklarationer för medlemsfunktionerna. Vi visar även hur en *gard*, som alltid ska finnas i en inkluderingsfil för att förhindra multipel inkludering, kan se ut:

```
#ifndef POINT_H
#define POINT_H

class Point {
public:
  Point(int x, int y);

  int getX() const;
  int getY() const;

  void setX(int x);
  void setY(int y);

private:
  int  x;
  int  y;
};

#endif
```

På en tillhörande *implementeringsfil* med namnet `Point.cc`, eller annat känt filnamnssuffix för C++, skrivs definitionerna för medlemsfunktionerna. För att ange att dessa funktionsdefinitioner avser medlemsfunktioner i klassen `Point`, anges klassnamnet ihop med räckviddsoperatorn `::` som prefix till funktionsnamnen:

```
Point::Point(int x, int y) : x(x), y(y) {
}

int Point::getX() const {
  return x;
}

int Point::getY() const {
  return y;
}

void Point::setX(int x) {
  this->x = x;
}

void Point::setY(int y) {
  this->x = x;
}
```

Utan klassprefixet skulle kompilatorn tolka funktionerna som fria funktioner. I detta fall skulle dock användningen av **const** och **this** leda till kompileringsfel, eftersom dessa begrepp ej är relevanta i fria funktioner.

Arv

I Java finns endast enkelt arv, dvs en klass kan bara ha en superklass. Behovet av att kunna ärva från fler än en superklass (vanligtvis för att ärva in funktionalitet från övriga klasser) har man löst med gränssnitt (interface). Dessa är i princip abstrakta klasser med enbart abstrakta metoder och utan instansvariabler. Någon direkt motsvarighet till gränssnitt finns ej i C++.

I C++ används vanligtvis beteckningen *basklass* i stället för superklass. C++ tillåter *multipelt arv*, dvs en klass kan ärva från två eller flera basklasser. Även s.k. *upprepat arv* är möjligt. Detta innebär att om en viss klass X har flera basklasser, och dessa i sin tur har en gemensam basklass A, kommer klass A:s medlemmar att ärvas flera gånger till klassen X. Multipelt och upprepat arv ger upphov till både logiska och tekniska svårigheter, som även om de är lösbara gärna undviks.

Det finns tre sätt att ärva från en basklass, **public**, **protected** eller **private**. Detta avser främst hur **public**-medlemmar i basklassen ska ärvas, dvs om de ska hamna i den härledda klassens **public**-, **protected**- eller **private**-del. De två senare varianterna, arv med skyddad resp. privat basklass, innebär att man kan ärva bort egenskaper, vilket egentligen strider mot arvets grundläggande idé. Vi begränsar oss i fortsättningen till enkelt, publikt arv, det naturliga arvet.

Nedan definieras tre klasser, `Person`, `Man` och `Woman` (inom datalogin är de typiska manliga resp. kvinnliga egenskaperna skägg resp. barn). `Person` definieras enligt följande:

```
class Person {
public:
  Person(const string& name, int age = 0) : _name(name), _age(age) {}

  virtual void say_hello() const {
    cout << "Jag heter " << _name << " och är " << _age << " år." << endl;
  }

  string  get_name() const { return _name;}
  int     get_age() const { return _age;}

protected:
  string  _name;

private:
  int _age;
};
```

Från `Person` härleder vi subklassen `Man` och lägger till funktionalitet för skägg:

```
class Man : public Person {
public:
  Man(const string& name, int age = 0, bool beard = false)
    : Person(name, age), _beard(beard) {}

  void grow_beard() { _beard = true;  }
  void shave()      { _beard = false; }
  bool  have_beard() const { return _beard;  }

protected:
  bool  _beard;
};
```

Subklassen `Woman` härleds också från `Person` och funktionalitet för barn läggs till:

```
class Woman : public Person {
public:
  Woman(const string& name, int age = 0, int children = 0)
    : Person(name, age), _children(children) {}

  virtual void say_hello() const {
    cout << _name << " heter jag och är " << get_age() << " år." << endl;
  }

  void add_child()               { ++_children; }
  int   how_many_children() const { return _children; }

protected:
  int _children;
};
```

Arv anges syntaktiskt med ett kolon efter basklassens namn och därefter anges vilken typ av arv som avses (**public**, **protected** eller **private**), i detta fall med publik basklass.

Någon direkt motsvarighet till Javas **super** finns inte i C++. Överföring av argument till en basklass' konstruktor görs med initierare i subklassens initieringlista, `Person(name, age)` efter kolon i `Man`- och `Woman`-klassernas konstruktorer.

Deklarationen med **virtual** av medlemsfunktionen `say_hello`() i klassen Person innebär att denna funktion kan ridas över i subklasser och att den kommer att vara en polymorf funktion. Den ärv till Man-klassen men rids över i Woman-klassen. Upprepningen av **virtual** i Woman-klassen är ej nödvändig.

Någon uttrycklig deklaration motsvarande **abstract** i Java för att ange att en klass är abstrakt finns ej. En abstrakt klass i C++ uppstår om en klass deklareras med en *rent virtuell* (pure virtual) medlemsfunktion, vilket innebär att funktionen saknar definition. Detta anges med "= 0" i deklarationen:

```
virtual void say_hello() const = 0;
```

Polymorfi

I Java gäller alltid polymorfi då instansmetoder anropas. I C++ är det mycket mer invecklat och en hel del ansvar läggs på programmeraren för få polymorfi, om man önskar det, och även att det blir korrekt.

En grundförutsättning för att polymorfi ska gälla är att klassobjekt refereras via pekare eller referenser. Om så inte är fallet gäller statisk bindning av medlemsfunktionsanrop och det finns förvisso då heller inget tvivel om vad variabler har för typ.

En annan förutsättning är att de medlemsfunktioner som ska kunna ridas över och alltså vara polymorfa, måste deklareras **virtual** redan i den klass där en sådan metod först införs i en klasshierarki. Utan **virtual** kommer deklaration av samma funktion längre ner i klasshierarkin enbart att innebära omdeklaration, och anrop kan komma att bjuda på överraskningar genom att bindningen av medlemsfunktionsanrop enbart beror på typen för den pekar- eller referensvariabel som refererar till ett objekt (statiskt typ), inte objektets egentliga typ (dynamisk typ).

Objektet `w` nedan är ett statiskt deklarerat objekt vars typ och därmed är det också statiskt bestämt vilka operationer som kan utföras på `w`.

```
Woman w("Stina");
w.say_hello();                  // say_hello binds statiskt, w typ är statiskt bestämd
Person* p = new Woman("Stina");
p->say_hello();                 // say_hello binds dynamiskt
```

Destruktorer måste deklareras **virtual** när vi har klasshierarkier och polymorfi kan förekomma. En regel är att så snart en klass har en virtuell medlemsfunktion ska dess destruktor vara virtuell. Ser man inte till att detta gäller kommer inte destruering att göras korrekt.

RTTI och dynamisk typomvandling

Runtime Type Identification, RTTI, avser att dynamiskt bestämma typen för objekt. I Java finns operatorn **instanceof**, som returnerar sant om dess vänstra operand kan typomvandlas till den typ som anges av dess högra operand. Uttrycklig typomvandling kan behöva göras för att legalisera metodanrop (vanligtvis typomvandling till en subtyp, s.k. down-casting). I Java kan vi t.ex. skriva följande:

```
if (obj instanceof Woman)
  n = ((Woman) obj).how_many_children();
```

I C++ finns en funktion `typeid`() som kan anropas för det objekt man vill kontrollera typen för och den returnerar ett objekt av typen `type_info`. Ett `type_info`-objekt kan jämföras med ett annat `type_info`-objekt med operatorn `==` och den returnerar sant om de två `type_info`-objekten är lika, dvs beskriver samma typ.

Typomvandling av pekare och referenser görs med typomvandlingsoperatorn **dynamic_cast**. Den kan ta den pekartyp (referenstyp) man vill omvandla till samt det pekarvärde (referensvärde) som ska typomvandlas. Följande kontrollerar först om pekaren `p` pekar på ett objekt av typen `Woman` (eller en subtyp till `Woman`). Om så är fallet typomvandlas `p` till `Woman*` och den `Woman`-specifika medlemsfunktionen `how_many_children()` anropas.

```
if (typeid(*p) == typeid(Woman))
  n = dynamic_cast<Woman*>(p)->how_many_children();
```

Typomvandlingsoperatorn **dynamic_cast** kan också användas för typkontroll. Om en pekare som man typomvandla inte pekar på ett objekt av den typ som pekartypen man vill omvandla till anger (eller en subtyp till denna), returnerar **dynamic_cast** en tompekare, dvs 0.

```
if (dynamic_cast<Man*>(p) != 0)
  dynamic_cast<Man*>(p)->shave();
```

Om en referens inte refererar till ett objekt av den typ man vill typomvandla referensen till, returneras inte en nolla, utan i detta fall genereras undantaget `bad_cast`.

Operatoröverlagring

I Java kan enbart metodnamn (identifierare) överlagras. I C++ kan även operatorer överlagras, antingen som fria funktioner eller som en medlemmar i klasser. Vissa operatorer måste vara medlem i en klass, andra kan inte vara det, men för de flesta kan man välja och rentav överlagra i båda formerna.

Alla operatorer i språket med några få undantag kan överlagras. Man kan inte hitta på nya operatorsymboler. Prioritet, associativitet och antal operander kan inte ändras för en operator när den överlagras, utan det som är definierad i språket gäller även egna överlagringar. Syntaxen för att namnge operatorer är det reserverade ordet **operator** tillsammans med operatorsymbolen, t.ex. **operator+**.

Då man överlagrar en operator som medlem i en klass får man hålla i minnet att ett klassobjekt som operatorn anropas för kommer att finnas som ett implicit argument. En tvåställig (binär) operator kommer därför enbart ha *en* parameter, som motsvarar högeroperanden. En enställig (unär) operator kommer inte att ha någon parameter. Om vi skriver anropet av en operator med vanlig punktnotation, vilket också är legalt, inser vi detta:

```
a.operator=(b);     är detsamma som     a = b;
i.operator++();     är detsamma som     ++i;
```

Möjligheten att t.ex. överlagra operatorn `<<` innebär att vi kan göra formaterad utmatning för egendefinierade typer, helt i överensstämmelse med vad som kan göras för inbyggda typer. För datatypen `Entry`, som använts i tidigare exempel, kan vi t.ex. överlagra `<<` enligt följande:

```
ostream& operator<<(ostream& os, const Entry& e) {
  os << e.id << ": ";
  switch (e.type) {
    case CHAR:   os << e.c;
    case INT:    os << e.i;
    case DOUBLE: os << e.d;
  }
  return os;
}
```

Detta tillåter utskrift av ett `Entry`-objekt `entry` enligt följande:

```
cout << entry << endl;
```

Eftersom operatorn `<<` har ett vänsterargument som är av typen `ostream&`, kan den inte vara medlem i `Entry` utan måste vara en fri funktion.

Ett exempel på överlagring av en operator som medlemsfunktions kan vara tilldelningsoperatorn, **operator**=, vilket också är ett exempel på en operator som måste vara medlem (detta för att garantera att vänsterargumentet är en variabel, ett *lvalue*). Tilldelningsoperatorn för inbyggda typer returnerar ett *lvalue*, så vi låter därför vår överlagring returnera en referens till objektet den anropats för:

```
struct Entry {
  Entry& operator=(const Entry&);

  string  id;
  Kind    type;
  union {
    char    c;
    int     i;
    double  d;
  };
};
```

I definitionen kontrollerar vi att inte samma objekt finns både till vänster och höger, genom att jämföra adressen för högerargumentet med **this**, adressen till vänsterargumentet. Detta vill vi göra framför allt för att inte göra olämpliga saker, men om det är samma objekt finns ju heller inget att göra.

```
Entry& Entry::operator=(const Entry& rhs)
{
  if (this != &rhs) {
    id = rhs.id;
    switch (rhs.type) {
      case CHAR:   c = rhs.c;
      case INT:    i = rhs.i;
      case DOUBLE: d = rhs.d;
    }
  }
  return *this;
}
```

Speciellt när vi har klasser med datamedlemmar som är pekare till dynamiskt minne är överlagring av tilldelningsoperatorn vanligtvis nödvändigt för att erhålla korrekt semantik (djup tilldelning/kopiering).

Undantag

I både Java och C++ finns undantag (**exception**) och likheterna är stora. **try**-block, **throw**-uttryck (sats i Java) och **catch**-klausuler är identiska med motsvarande i Java. Den **finally**-klausul som kan tillfogas **try**-blocken i Java finns dock inte i C++. I Java skapas alltid undantagsobjekt med **new**, medan man i C++ *inte* gör det. Klassobjekt i C++ behöver inte skapas dynamiskt och det är inte heller så det görs i de fall standardbiblioteksprogram genererar undantag.

C++ har en uppsättning standardundantag definierade i form av en klasshierarki, med klassen `exception` som rotklass. Undantagen delas sedan upp i huvudkategorierna `logic_error` och `runtime_error`, och under dessa kommer mer specifika undantagsklasser som t.ex. `out_of_range`, `length_error` etc.

Användningen av undantag i språken skiljer påtagligt. Java använder undantag mycket mer och programmeraren tvingas i mycket högre grad att hantera undantag, på något sätt. I C++ är användningen mer begränsad och det är dessutom upp till programmeraren om undantag ska hanteras eller ej (gör man inte det kraschar förstås programmet till slut). En funktion som kan generera undantag behöver t.ex. inte deklarera detta genom en **throw**-klausul i funktionsdeklarationen:

```
int find(int pos) {
  if ( … ) throw out_of_range(”Otillåten position”);
  …
}
```

Följande exempel visar hur undantagshanteringen ser ut i C++. Konstantreferenser används normalt för att deklarera undantag i **catch**-klausuler.

```
try {
  …
  int  value = find(pos);
  …
}
catch (const logic_error& e) {
  cout << "Logiskt fel: ” << e.what() << endl;
}
catch (const exception& e) {
  cout << typeid(e).name() << ”: ” << e.what() << endl;
  throw;
}
catch (...) {
  cout << ”Okänt fel har inträffat!” << endl;
  exit(1);
}
```

Undantag av typen `logic_error` fångas in i den första **catch**-klausulen. I den andra **catch**-klausulen fångas alla andra undantag av typen **exception** in. Det enkla **throw**-uttrycket kastar samma undantag vidare. Den sista **catch**-klausulen använder *ellipsnotation* (...) för att fånga in undantag som inte fångas av de två första **catch**-klausulen. Det är i så fall ett undantag som inte tillhör standardundantagen i `exception`-hierarkin. Funktionen `typeid`() returerar ett objekt av typen `type_info`, som beskriver typen för det argument som ges, `e`. Medlemsfunktionen `name`() returnerar namnet på typen för som en sträng, i detta fall namnet på den undantagsklass som `e` är en instans av. Funktionen `exit`() är en standardfunktion motsvarande `System.exit`() i Java.

Trådar

Någon motsvarighet till trådar (*threads*) finns inte i C++, utan i C++ förlitar man sig helt på operativsystemanrop för att skriva program med samtidighet. I C++09 (arbetsnamn för den standard som är planerad att ersätta den gamla 2009) kommer troligtvis trådhantering att införas.

Namnrymder

En namnrymd är en mekanism för att logiskt gruppera sammanhörande deklarationer. Ett exempel:

```
namespace Lexical_Analyzer {
  enum Token {
    NAME, NUMBER, END = ';', LPAR = '(', RPAR = ')',
    PLUS = '+', MINUS = '-', MUL = '*', DIV = '/', ASSIGN = '='
  };
```

```
  Token   current_token;
  double  number;
  string  sval;

  Token get_token() { … }
}
```

Funktionen `get_token`() skulle också enbart kunna deklareras i namnrymden och definieras separat enligt följande:

```
Token Lexical_Analyzer::get_token() { … }
```

För att komma åt ett namn i en namnrymd kan man använda dess kvalificerade namn, t.ex.

```
if (Lexical_Analyzer::current_token == Lexical_Analyzer::PLUS) …
```

Man kan öppna hela namnrymden med ett **using**-direktiv. Namnen i namnrymden blir då direkt synliga:

```
using namespace Lexical_Analyzer;

if (current_token == PLUS) …
```

Vill man kunna använda ett enkelt namn utan att öppna hela namnrymden kan man göra en **using**-deklaration som endast avser det namnet:

```
using Lexical_Analyzer::current_token;

if (current_token == Lexical_Analyzer::PLUS) …
```

Ovanstående visar enbart de grundläggande egenskaperna hos namnrymder. Det finns fler möjligheter att utforma och använda namnrymder, bl.a. anonyma namnrymder för att begränsa namns räckvidd till en viss kompileringsenhet.

Mallar

Mallar (templates) finns inte i Java. Det ansågs som en källa till problem och det finns visst fog för den åsikten. Å andra sidan måste man konstatera att mallar med stor framgång har använts i flera språk, bl.a. Ada och C++. T.ex. är stora delar av standardbiblioteket i C++ baserat på klass- och funktionsmallar och det är generellt, utvidgbart och effektivt. Somliga tror att mallar med tiden kommer att införas även i Java och det finns redan utvidgade versioner av Java som omfattar mallar.

I grunden används mallar för att parametrisera med avseende på *datatyp*, ungefär på samma sätt som man använder funktionsparametrar för att parametrisera med avseende på *objekt*. En funktion kan ju t.ex. vid olika tillfällen anropas med olika argument och på så sätt återanvändas för *olika objekt* av *samma typ*. En ytterligare generalisering vore att även kunna variera typen för en funktions parametrar.

För många slags datastrukturer är det typiskt att de skulle kunna återanvändas för element av olika typ. Behållarklasserna i C++:s standardbibliotek, `list`, `vector`, `map`, `set`, osv. är bra exempel på detta. Även många vanliga funktioner skulle kunna återanvändas för olika parametertyper. Ett sådant exempel är funktionen för absolutbelopp, `|a|`, i en **int**-version:

```
int abs(int a) {
  return (a < 0) ? -a : a;
}
```

Denna funktion skulle kunna användas både för heltal och flyttal om parametertypen som är densamma som returtypen kunde parametriseras ut, vilket den kan:

```
template<class T>
T abs(T a) {
  return (a < 0) ? -a : a;
}
```

Mallprefixet **template**<…> anger att det följer en malldeklaration. Mellan < och > deklareras mallparametrarna, i detta fall en parameter **class** T, som är en formell typparameter. **class** är i detta sammanhang säger enbart att T är en typparameter och innebär *inte* att T ska vara av klasstyp. Man kan i stället för **class** i detta sammanhang använda **typename**, vilket tydligare anger vad det är fråga om. I resten av funktionsdefinitionen används typparametern T där vi senare vill använda en konkret typ.

För att kunna använda funktionsmallen måste den instansieras med en konkret typ. Det kan göras genom att helt enkelt skriva ett funktionsanrop, t.ex.

```
double  x = -21;
cout << abs(x) << endl;
```

Ett sådant anrop innebär att en instans av mallen instansieras och den formella typen `T` ersätts med den konkreta **double**. Detta är inte helt okomplicerat och det finns ett rigoröst regelverk för hur kompilatorn ska gå tillväga för att säkert hantera funktionsanrop, med tanke på förekomst av vanliga funktioner, funktionsmallar, redan instansierade funktionsmallar och möjligheten att automatiskt typomvandla funktionsargument.

Den `Array`-klass vi använt i tidigare exempel skulle kunna vara en lämplig klass att definiera som en klassmall, genom att parametrisera ut elementtypen:

```
template<class T>
class Array {
public:
  Array(int size = 100) : size_(size), element_(new T[size]) {}
  ~Array() { delete[] element; }

  …
private:
  int  size_;
  T*   element_;
};
```

Användning av klassmallen:

```
Array<double>  double_vector(200);
```

Typen för `double_vector` är `Array<double>`. Detta är typmässigt skilt från t.ex. en **int**-instans, `Array<int>`, vilket innebär att stark typning gäller för mallar som instansierats med olika mall-argument, typer i detta fall.

Standardbibliotek

Både Java och C++ har omfattande standardbibliotek. Det finns stora innehållsmässiga skillnader men även då det gäller bibliotekens konstruktion är skillnaden stor.

Javas standardbibliotek är baserat på ren objektorienterad teknik med en enhetlig och sammanhållen klasshierarki, där klassen `Object` är alla klassers gemensamma basklass. Paket används för att dela upp klasserna i olika kategorier.

I Java används polymorfa objektreferenser, dynamisk bindning och uttrycklig dynamisk typomvandling för att uppnå generalitet och återanvändbarhet.

I Javas standardbibliotek finns hantering av grafik, nätverkskommunikation och mycket annat, som saknas i standardbiblioteket för C++.

C++ standardbibliotek

Standardbiblioteket för C++ är inte lika enhetligt som Javas. Det beror mycket på att standardbiblioteket för C har införlivats och att därutöver omfattande C++-specifika tillägg gjorts. Detta har medfört en blandning av gammal C-teknologi och modernare konstruktioner. Dessutom har det uppkommit en hel del överlapp. Det finns t.ex. två olika sätt att göra in- och utmatning, två olika typer för att hantera strängar och två olika sätt att hantera dynamiskt minne, bara för att ge exempel på några överlappningar.

De C++-specifika delarna av biblioteket är huvudsakligen baserade på kodmallar, både då det gäller datatyper som funktionalitet. Mycket av funktionaliteten bygger på iteratorer, vilka motsvaras av gränssnitten `Iterator` och `Enumeration` i Java.

Tekniken med kodmallar, vilka under kompileringen eller länkningen instansieras för specifika datatyper, etc., innebär att datatyper binds statiskt och följaktligen kan kontrolleras statiskt. Detta eliminerar behovet av dynamiska kontroller, vilket gör programmen effektiva.

C++ har behållit en hel del av standardbibliotek för C, där in- och utmatning, matematiska funktioner, dynamisk minneshantering, tecken- och stränghantering (C-strängar), och mycket annat ingår. Därutöver har omfattande tillägg gjort, som dels är unika, dels (vanligtvis) bättre alternativ till vad som redan ingår i C-biblioteket.

Följande är en kort översikt av vad standardbiblioteket innehåller som är specifikt för C++. Namn som skrivits med `Courier` är i de flesta fall namn på inkluderingar som ska göras för att få tillgång till namnen ifråga och i de flesta fall namn på datatyper (men det finns undantag).

- elva *standardbehållare*: `vector`, `list`, `deque`, `queue`, `priority_queue`, `stack`, `map`, `multimap`, `set`, `multiset` och `bitset`. (`multimap`, `multiset` och `priority_queue` deklarerar ihop med `map`, `set` resp. `queue`).
- *iteratorer* (`iterator`) för att genomlöpa behållare. Varje klass som använder iteratorer har medlemsfunktioner för att returnera iteratorer som pekar in i datastrukturerna.
- *algoritmer* (`algorithm`) för att operera på behållare.
- en del allmäna hjälpmedel som t.ex. vissa generella operatorer och en partyp (`utility`), funktionsobjekt (`functional`) som används för att parametrisera funktionalitet i klassmallar, minnestilldelning för behållare (`memory`).
- *undantag* (`exception`, `stdexcept`) för att generera, identifiera och hantera fel, etc.
- en riktig *strängtyp* (`string`) med strängobjekt som beter sig som första klassens objekt (sköter sitt minne själv; tilldelning, jämförelser, etc. fungerar som sig bör).
- ett *strömbiblioteket* (`iostream`, `iomanip`) för in- och utmatning, filhantering (`fstream`), etc.
- *lokalisering* (`locale`), dvs igenkänning och hantering av nationella skillnader, t.ex. med avseende på datumformat, valutasymboler, bokstavsuppsättningar, etc.
- programmeringstöd, t.ex. numeriska gränser (`limits`), dynamisk minneshantering (`new`), dynamisk typidentifiering (`typeinfo`), undantag (`exception`).
- numeriska funktioner av många slag (`complex`, `valarray`, `numerics`).

Utöver ovanstående finns alltså mer i form av C-baserad funktionalitet.

Appendix A: Inbyggda datatyper

bool	booletyp (**false** eller **true**)
[**unsigned** | **signed**] **char**	teckentyp, med eller utan teckenbit
[**unsigned** | **signed**] **short** [**int**] [**unsigned** | **signed**] **int** [**unsigned** | **signed**] **long** [**int**]	heltalstyper av olika storlek, med eller utan förtecken
float **double** **long double**	flyttalstyper av olika storlek
void	används som returtyp för funktion som ej returnerar ett värde, eller som bastyp för pekare till objekt av okänd typ (**void***)

Appendix B: Användardefinierade datatyper

enum *identifierare* { *identifierarlista* };	Uppräkningstyp
*T**	Pekartyp (pekare till objekt av typ T)
T&	Referenstyp (referens till objekt av typ T)
T[ ]	Fälttyp; för flerdimensionella fält anges ett klammerpar för varje dimension (definieras alltså som fält av fält)
struct *identifierare* { *medlemsspecifikationer* };	*Structure*; förvalt åtkomstskydd **public** (överensstämmer f.ö. med **class**)
class *identifierare* { *medlemsspecifikationer* };	Klass
union *identifierare* { *medlemsspecifikationer* };	Unionstyp; gemensamt minnesutrymme för samtliga medlemmar, endast ett värde i taget kan lagras.

Appendix C: Operatoröversikt

Tabellen nedan visar operatorerna i C++, ordnade i sjunkande *prioritet*. Kolumnen längst till höger anger *associativiteten* för respektive operator, $\mathcal{V}$ anger vänsterassociativitet, $\mathcal{H}$ anger högerassociativitet.

Operator	**Funktion**	**Assoc**
klassnamn `::` *medlem*	klassmedlem	$\mathcal{V}$
namnrymdsnamn `::` *medlem*	namnrymdsmedlem	$\mathcal{V}$
`::` *namn*	globalt namn	$\mathcal{H}$
`::` *kvalificerat-namn*	globalt namn	$\mathcal{H}$
objekt `.` *medlem*	medlemselektion i **struct** eller **class**	$\mathcal{V}$
pekare `->` *medlem*	medlemselektion i **struct** eller **class**	$\mathcal{V}$
pekare `[` *uttryck* `]`	indexering av fält	$\mathcal{V}$
uttryck `(` *uttryckslista* `)`	funktionsanrop	$\mathcal{V}$
enkel-typ `(` *uttryckslista* `)`	värdekonstruktion (typomvandling)	$\mathcal{V}$
lvalue `++`	postfix uppstegning	$\mathcal{H}$
lvalue `--`	postfix nedstegning	$\mathcal{H}$
typeid `(` *type* `)`	typidentifiering	$\mathcal{H}$
typeid `(` *uttryck* `)`	dynamisk typidentifiering	$\mathcal{H}$
dynamic_cast`<`*typ*`>(`*uttryck*`)`	dynamisk typomvandling	$\mathcal{H}$
static_cast`<`*typ*`>(`*uttryck*`)`	statisk typomvandling	$\mathcal{H}$
reinterpret_cast`<`*typ*`>(`*uttryck*`)`	okontrollerad typomvandling	$\mathcal{H}$
const_cast`<`*typ*`>(`*uttryck*`)`	konstantomvandling	$\mathcal{H}$
sizeof *uttryck*	objektstorlek	$\mathcal{H}$
sizeof `(` *typ* `)`	typstorlek	$\mathcal{H}$
`++` *lvalue*	prefix uppstegning	$\mathcal{H}$
`--` *lvalue*	prefix nedstegning	$\mathcal{H}$
`~` *uttryck*	komplement	$\mathcal{H}$
`!` *uttryck*	logisk negation (icke)	$\mathcal{H}$
`+` *uttryck*	enställigt (unärt) plus	$\mathcal{H}$
`-` *uttryck*	enställigt (unärt) minus	$\mathcal{H}$
`&` *lvalue*	adresstagning	$\mathcal{H}$
`*` *uttryck*	avreferering av pekare	$\mathcal{H}$
new *typ*	tilldela dynamiskt minne	$\mathcal{H}$
new *typ* `(` *uttrycklista* `)`	tilldela dynamiskt minne och initiera	$\mathcal{H}$
new `(` *uttryckslista* `)` *typ*	*placement syntax*; extra arg. till **new**	$\mathcal{H}$
new `(` *uttryckslista* `)` *typ* `(` *uttryckslista* `)`	*placement syntax* och initiering	$\mathcal{H}$
delete *pekare*	återlämna dynamiskt minne	$\mathcal{H}$
delete `[ ]` *pekare*	återlämna dynamiskt minne för fält	$\mathcal{H}$
`(` *typ* `)` *uttryck*	statisk typomvandling (”cast”)	$\mathcal{H}$
objekt `.*` *pekare-till medlem*	pekare-till-medlem-selektion	$\mathcal{V}$
pekare `->*` *pekare-till medlem*	pekare-till-medlem-selektion	$\mathcal{V}$

Operator	**Funktion**	**Assoc**
uttryck `*` *uttryck*	multiplikation	$\mathcal{V}$
uttryck `/` *uttryck*	division	$\mathcal{V}$
uttryck `%` *uttryck*	modulo (rest vid heltalsdivision)	$\mathcal{V}$
uttryck `+` *uttryck*	addition	$\mathcal{V}$
uttryck `-` *uttryck*	subtraktion	$\mathcal{V}$
uttryck `<<` *uttryck*	bitvis vänsterskift (nollutfyllnad)	$\mathcal{V}$
uttryck `>>` *uttryck*	bitvis högerskift (aritmetiskt, logiskt?)	$\mathcal{V}$
uttryck `<` *uttryck*	mindre än	$\mathcal{V}$
uttryck `<=` *uttryck*	mindre än eller lika med	$\mathcal{V}$
uttryck `>` *uttryck*	större än	$\mathcal{V}$
uttryck `>=` *uttryck*	större än eller lika med	$\mathcal{V}$
uttryck `==` *uttryck*	lika med	$\mathcal{V}$
uttryck `!=` *uttryck*	ej lika med	$\mathcal{V}$
uttryck `&` *uttryck*	bitvis OCH	$\mathcal{V}$
uttryck `^` *uttryck*	bitvis exklusivt ELLER	$\mathcal{V}$
uttryck `|` *uttryck*	bitvis (inklusivt) ELLER	$\mathcal{V}$
uttryck `&&` *uttryck*	logiskt OCH	$\mathcal{V}$
uttryck `||` *uttryck*	logiskt ELLER	$\mathcal{V}$
lvalue `=` *uttryck*	enkel tilldelning	$\mathcal{H}$
lvalue `*=` *uttryck*	multiplicera och tilldela	$\mathcal{H}$
lvalue `/=` *uttryck*	dividera och tilldela	$\mathcal{H}$
lvalue `%=` *uttryck*	modulo och tilldela	$\mathcal{H}$
lvalue `+=` *uttryck*	addera och tilldela	$\mathcal{H}$
lvalue `-=` *uttryck*	subtrahera och tilldela	$\mathcal{H}$
lvalue `<<=` *uttryck*	skifta vänster och tilldela	$\mathcal{H}$
lvalue `>>=` *uttryck*	skifta höger och tilldela	$\mathcal{H}$
lvalue `&=` *uttryck*	bitvis OCH och tilldela	$\mathcal{H}$
lvalue `^=` *uttryck*	bitvis exklusivt ELLER och tilldela	$\mathcal{H}$
lvalue `|=` *uttryck*	bitvis (inklusivt) ELLER och tilldela	$\mathcal{H}$
uttryck `?` *uttryck* `:` *uttryck*	villkorsuttryck	$\mathcal{V}$
throw *uttryck*	*throw*-uttryck (har typen **void**)	
uttryck`,` *uttryck*	kommauttryck	$\mathcal{V}$

Appendix D: Satsöversikt

Nedan följer syntaxen för satserna i C++. Beskrivningen är i viss mån förenklad och inte heller fullständigt beskriven i detalj. Syntaktiska element markerade med $_{opt}$ är valfria och kan utelämnas.

Syntax	**Kommentar**
sats :	en sats avslutas alltid av ett semikolon
etiketterad-sats	sats med någon form av prefix
uttryckssats	ett uttryck följt av semikolon blir en sats
sammansatt-sats	även kallad *blocksats*
selektionssats	val
iterationssats	även kallad *repetitionssats*
hoppsats	bryter den sekventiella exekveringen
deklarationssats	även deklarationer räknas som satser
try-block	för infångande av undantag
etiketterad-sats :	
identifierare : *sats*	enda användning är som mål för **goto**
case *konstant-uttryck* : *sats*	läge i **switch**
default : *sats*	valfritt, sista läge i **switch**
uttryckssats :	
$uttryck_{opt}$;	enbart semikolon kallas *tom sats*
sammansatt-sats :	satsblock, vanligtvis bestående av flera satser
{ $satssekvens_{opt}$ }	blockparentes, kan vara tom.
selektionssats :	
if (*villkor*) *sats*	*sats* utförs om *villkor* beräknas till SANT
if (*villkor*) *sats* **else** *sats*	*sats* efter **else** utförs då *villkor* FALSKT
switch (*villkor*) *sats*	*sats* är normalt en sammansatt sats
villkor :	*Integral* el. **enum** för **switch**, övriga **bool**
uttryck	av typ **bool**, *integral* eller **enum**
typspecificerare deklarator = *uttryck*	räckvidden är de undersatser *villkor* styr över
iterationssats :	
while (*villkor*) *sats*	förtestad, villkorsstyrd repetition
do *sats* **while** (*villkor*) ;	eftertestad, villkorsstyrd repetition
for (*for-initieringssats* $villkor_{opt}$; $uttryck_{opt}$) *sats*	förtestad, villkorsstyrd repetition
for-initieringssats :	*anm*. Avslutas med semikolon
uttryckssats	t.ex. i = 0;
enkel-deklaration	t.ex. **int** i = 0;
hoppsats :	
break ;	hopp ur iterationssats eller **switch**
continue ;	hopp till nästa iteration i iterationssats
return $uttryck_{opt}$;	retur från funktion, *uttryck* anger returvärde
goto *identifierare* ;	hopp till etiketterad sats

deklarationssats :	
blockdeklaration	deklaration som kan förekomma i block.
blockdeklaration :	
enkel-deklaration	t.ex. en variabeldeklaration
asm-deklaration	**asm** (*stränglitteral*) ;
namnrymdalias-deklaration	**namespace** *identifierare* = …
using-deklaration	**using** *typnamn* :: *identifierare*
using-direktiv	**using namespace** …
try-block : [a]	
try { *satssekvens*$_{opt}$ } **catch** (*undantagsdeklaration*) { *satssekvens*$_{opt}$ }	
try *sammansatt-sats hanterarsekvens*	mer formell syntax än ovanstående exempel.
hanterarsekvens :	
hanterare hanterarsekvens$_{opt}$	
hanterare :	
catch (*undantagsdeklaration*) *sammansatt-sats*	kan finnas en eller flera till ett **try**-block
undantagsdeklaration :	
typspecifierare deklarator	t.ex. **const** exception& e
typspecifierare	t.ex. **const** exception&
...	endast som sista hanterare, fångar allt.

a. Det finns även en konstruktion som kallas *funktion-try-block*, som kan förekomma i en konstruktor och där används för att fånga undantag som genereras i konstruktorns *initierare*. Det är dock ingen sats.

Litteratur

[1] Bjarne Stroustrup: The Design and Evolution of C++. Addison Wesley, 1994.

[2] Bjarne Stroustrup: The C++ Programming Language. Third Edition. Addison-Wesley, 1997.

[3] International Standard for Information Systems – Programming Language C++. ISO/IEC 14882, 1998.