C++-manual

Jan Ekholm

chakie@infa.abo.fi

C++-manual
av Jan Ekholm

Copyright © 1999, 2000 av Jan Ekholm

Ett kompendium skapat för att fungera som bas för undervisning i kursen *C++-programmering*. Detta kompendium är dock ännu under arbete och mycket kan ännu ändras!

Denna text får fritt kopieras enligt *GNU General Public License* (GPL).

Revisions Historik;
Revision 1.42000-02-23

Innehållsförteckning

1. Introduktion 15
1.1. Vad är C++ 15
1.1.1. Historia 15
1.1.2. C och C++ 16
1.2. Om detta kompendium 16
1.2.1. Copyright 17
1.2.2. Notation 17
1.3. Ett första program 18
1.3.1. Hello world! 18
1.3.2. Konventioner för namngivning av filer 19
1.3.3. Kompilera och köra program 20
1.4. Grundläggande syntax 20
1.4.1. Separering av satser 21
1.4.2. Kodblock 21
1.4.3. Formatering av kod 21
1.4.4. Kommentarer 22
2. Variabler och datatyper 24
2.1. Primitiva datatyper 24
2.1.1. Heltal 24
2.1.1.1. Talbaser 25
2.1.2. Flyttal 25
2.1.3. Tecken 26
2.1.4. Strängar 26
2.1.5. Booleans 27
2.2. Deklarera variabler 27
2.2.1. Giltiga variabelnamn 28
2.3. Tilldelning av variabler 29
2.3.1. Typkonvertering 30
2.4. Konstanter 31
3. Aritmetik 33
3.1. Normala aritmetiska operatorer 33

3.1.1. Jämföra variabler 33
3.1.2. Precedens 34
3.1.3. Modulus 34
3.2. Mera om division 35
3.2.1. Heltalsdivision 35
3.2.2. Flyttalsdivision 35
3.2.3. Precisionsproblem 36
3.2.4. Division med `0` 36
3.3. Binär aritmetik 37
3.3.1. Shiftoperatorerna « och » 37
3.3.2. Bitvisa operatorerna `|`, `&`, `^` och `~` 38
3.4. Inkrementerings- och dekrementeringsoperatorerna `++` och `--` 39
3.5. Sammansatta tilldelningar 40

4. Enkel input/output 42

4.1. Utskrift till skärmen 42
4.2. Inmatning från tangentbordet 42
4.3. Headerfilen `iostream` 43

5. Iteration 44

5.1. `while`-slingor 44
5.1.1. Eviga slingor 45
5.1.2. Nästlade slingor 46
5.2. `do-while`-slingor 46
5.3. `for`-slingor 47
5.4. Avbryta och fortsätta slingor 49
5.4.1. Avbrytning med `break` 49
5.4.2. Fortsättning med `continue` 50

6. Flödeskontroll 52

6.1. `if`-satser 52
6.1.1. Gruppering av kod i block 53
6.1.2. Nästlade `if`-satser 54
6.2. `if-else`-satser 54
6.3. Logiska operatorer 56
6.3.1. Lazy evaluation 57

6.3.2. Kombinera uttryck ... 58
6.4. `switch`-uttryck ... 58
6.5. `? :`-operatorn ... 61
7. Funktioner ... 63
7.1. Grundbegrepp ... 63
7.1.1. Parametrar till funktioner ... 63
7.1.2. Returvärden ... 64
7.2. Anropa funktioner ... 66
7.3. Parametertyper ... 66
7.3.1. Referensparametrar ... 67
7.3.2. Konstanta parametrar ... 68
7.4. Funktionen `main()` ... 70
7.4.1. Returvärde från `main()` ... 71
7.4.2. Prototyper ... 72
7.5. Parametrar till `main()` ... 73
7.6. Omfattning (scoping) ... 76
7.6.1. Lokala variabler ... 77
7.6.2. Globala variabler ... 77
7.6.3. Statiska variabler ... 78
7.7. Rekursion ... 80
7.8. Funktioner som parametrar ... 82
7.8.1. Funktionsvariabler ... 82
7.8.2. Anropa funktionsvariabler ... 83
8. Input och output ... 86
8.1. Streams och buffring ... 86
8.1.1. Buffring av data ... 86
8.2. Utskrift till skärmen ... 87
8.2.1. Buffring och `cout` ... 88
8.2.2. Utskrift i olika talbaser ... 89
8.2.3. Utskrift av flyttal ... 91
8.2.4. Formatering av utskrifter ... 93
8.3. Inmatning från tangentbordet ... 97
8.4. Utskrift till `cerr` och `clog` ... 100

9. Avancerade datatyper ... 101
9.1. Pekare ... 101
9.1.1. Avreferera pekare ... 104
9.1.2. Pekare som parametrar ... 105
9.1.3. Konstanta pekare ... 106
9.2. Vektorer ... 107
9.2.1. Indexering av vektorer ... 109
9.2.2. Tilldelning och jämförelse av vektorer ... 110
9.3. Vektorer som pekare ... 110
9.3.1. Pekararitmetik ... 111
9.3.2. Vektorer som funktionsparametrar ... 113
9.4. Flerdimensionella vektorer ... 114
9.4.1. Flerdimensionella vektorer och minne ... 115
9.5. Sammansatta datatyper ... 116
9.5.1. Scoping ... 117
9.5.2. Accessera medlemmar ... 119
9.5.3. Jämföra och tilldela ... 119
9.5.4. Sammansatta datyper som parametrar ... 120
9.5.4.1. Effektivitet vid parameteröverföring ... 122
9.6. Enumereringar ... 123
9.7. Definiera egna datatyper ... 125
10. Preprocessorn ... 127
10.1. Vad är en preprocessor? ... 127
10.2. Inkludera filer ... 127
10.2.1. Standarden för C++ ... 128
10.3. Konstanter ... 129
10.4. Makron ... 130
10.4.1. Nackdelar med makron ... 131
10.5. Multipel inkludering ... 132
11. Externa bibliotek ... 135
11.1. Vad är bibliotek? ... 135
11.1.1. Bibliotek under Unix ... 135
11.1.2. Bibliotek under Windows ... 135

11.2. Hur bibliotek fungerar....136
11.2.1. Bibliotek och header-filer....137
11.3. Fördelen med bibliotek....137
11.4. Exempelbibliotek....138
11.4.1. Matematikbibliotek....139
11.4.2. X11 och Windows....139
11.4.3. OpenGL....139
12. Dynamisk minneshantering....140
12.1. Vad är dynamisk minneshantering....140
12.2. Allokering och frigöring av minne....140
12.2.1. Allokering av minne....140
12.2.2. Allokera vektorer....141
12.2.3. Allokering av sammansatta datatyper....143
12.2.4. Frigöring av minne....146
12.3. Vanliga fel....148
12.3.1. Onitialiserat minne....149
12.3.2. Frigjort minne....149
12.3.3. Dubbel frigöring av minne....150
12.3.4. Fel indexering av vektor....150
12.3.5. Minnesläckor....151
12.4. Minneshantering på C:s vis....152
13. Kompilera C++-program....155
13.1. Kompilatorer....155
13.1.1. Kompilera program....156
13.1.2. Optimering och debuggning....157
13.1.3. Varningar....158
13.1.4. Sökstig för headerfiler....160
13.2. Länkning av program....160
13.2.1. Sökstig för bibliotek....161
13.2.2. Biblioteksberoenden....162
13.3. Använda multipla filer....162
13.3.1. Exempel på separat kompilering....164
14. Klasser....168

14.1. Allmänt om objektorientering....168
14.1.1. Abstraktion....168
14.1.2. Återanvändning av kod....169
14.1.3. Andra fördelar med OO i C++....169
14.1.4. Objektorienterad design....170
14.2. En första klass....170
14.2.1. Dataskydd....172
14.3. Konstruktor och destruktor....174
14.3.1. Initialisering av medlemmar....175
14.3.2. Multipla konstruktorer....175
14.3.3. *Copy*-konstruktor....177
14.3.4. Destruktor....180
14.3.5. Mera om copy-konstruktor....182
14.4. Nästlade klasser....184
14.4.1. Klasser i klasser....185
15. Ärvning....187
15.1. Mål med ärvning....187
15.1.1. Relationer mellan objekt....187
15.1.2. Återanvändning av kod....188
15.1.3. Terminologi....189
15.2. Konkret exempel....189
15.2.1. Subklasser och konstruktorer....191
15.2.2. Subklasser och destruktorer....192
15.2.3. Anropa subklassers metoder....192
15.2.4. Diskussion....193
15.3. Överlagring av metoder....194
15.3.1. Virtuella metoder....194
15.3.2. Privata metoder....197
15.3.3. Dynamisk bindning....198
15.3.3.1. Dynamisk bindning och destruktorer....200
15.3.4. Abstrakta klasser....201
15.4. Typer av ärvning....202
15.4.1. `public` ärvning....202

15.4.2. `private` ärvning 202
15.4.3. `protected` ärvning 203
15.4.4. Sammanfattning 204
15.5. Referera till basklasser 205
16. Multipel ärvning 207
16.1. Vad är multipel ärvning 207
16.1.1. Problem med multipel ärvning 208
16.1.2. Antalet objekt 209
16.1.3. Vilken metod? 210
16.1.4. Diskussion 211
17. Mera om klasser 212
17.1. Dynamiskt allokera objekt 212
17.1.1. Exempel på dynamisk minneshantering 213
17.2. Friends 216
17.2.1. Friend-metoder 219
17.3. Polymorfism 219
17.3.1. Funktions- och metodpolymorfism 220
17.4. Standardvärden för funktioner och metoder 221
17.5. Statiska metoder och medlemmar 223
17.5.1. Statiska metoder 226
17.5.2. Singleton 227
17.6. Pekaren `this` 227
18. Stränghantering 230
18.1. Vad är `string` egentligen 230
18.2. Skapa strängar 230
18.2.1. Längden av strängar 231
18.3. Accessera enskilda tecken 232
18.4. Tilldelning och jämförelse 233
18.5. Insättning av text 234
18.5.1. Addera strängar 235
18.6. Söka text i strängar 236
18.6.1. Sökning från början av strängen 236
18.6.2. Sökning från slutet av strängen 237

18.6.3. Sökning av första och sista förekomst av tecken 237
18.6.4. Sökning av första och sista förekomst av tecken inte i söksträngen 238
18.7. Ersätta text 238
18.7.1. Radera text 239
18.8. Substrängar 239
18.9. Streams och strängar 240
18.9.1. Skriva till strängar 241
18.9.2. Läsa från strängar 243
18.9.3. Strängstreams och kompatibilitet 245
18.10. Manipulera tecken 246
18.10.1. Konvertera tecken 248
18.11. Konvertera till C-strängar 249
18.12. Iteratorer och `string` 250
19. Filhantering 252
19.1. Klasser 252
19.2. Läsa från en fil 252
19.2.1. Avsluta inläsning 254
19.2.2. Läsa teckenvis 255
19.2.3. Läsa radvis 257
19.2.4. Ignorera tecken 258
19.2.5. Läsa binär data 259
19.3. Skriva data till fil 260
19.3.1. Skriva teckenvis 262
19.3.2. Skriva binär data 263
19.4. Hantera felsituationer 263
19.5. Stänga en fil 265
20. Exceptions 266
20.1. Vad är *exceptions* 266
20.2. Använda exceptions 267
20.2.1. Allokering och exceptions 269
20.2.2. Exceptions och konstruktorer 270
20.2.3. Återkasta en exception 271
20.3. Ofångade exceptions 272

20.4. Hierarkier med exceptions 274
20.4.1. Ärvda exceptions 274
21. Överlagring av operatorer 277
21.1. Vad är *överlagring av operatorer*?..277
21.1.1. Operatorer som kan överlagras 277
21.1.2. Exempelklassen `Vector`..278
21.2. Överlagring av operatorn + ...280
21.2.1. Använda överlagrade operatorer 281
21.2.2. Överlagring av += 283
21.3. Överlagring av operatorn - 283
21.4. Överlagring och friends 285
21.4.1. Friend-funktioner 287
21.5. Operatorerna == och = ...288
21.5.1. Tilldelningsoperatorn = ...290
21.6. Överlagring av « och » 292
21.6.1. Överlagrad input med » 293
21.7. Överlagring av typkonverteringar 295
21.8. Diskussion..295
22. Serialisering 297
23. Typparametrisering 298
23.1. Iden med templates 298
23.2. Ett konkret exempel 300
23.2.1. Instantiering av templateklasser..302
23.3. Multipla parametrar 303
23.3.1. Definiera lättare namn..304
24. Standard Template Library 306
24.1. Vad är STL 306
24.1.1. Grundblock i STL 307
24.1.2. Effektivitet 307
24.2. Containers 308
24.2.1. Sekvenscontainers..308
24.2.2. Sorterade associativa containers 308
24.2.3. Instättning och access av data 309

24.2.4. Radering av data 311
24.2.5. Övrig funktionalitet hos containers....313
24.3. Iteratorer....313
24.3.1. Använda iteratorer....314
24.4. Generiska algoritmer....315
24.5. Adaptorer 316
24.6. Diskussion....316
25. Namespaces....318
25.1. Vad är ett *namespace* 318
25.1.1. Exempel på namnkollision....318
25.1.2. Namespacet `std` 319
25.2. Använda namespaces 319
25.2.1. Alltid använda ett namespace 320
25.2.2. Använda subset av namespace 321
25.2.3. Klasser som namespaces....322
25.3. Skapa egna namespaces 322
25.3.1. Namespaces och prototyper 324
25.4. Aliaser....327
25.5. Diskussion....328
26. Typinformation....329
26.1. Varför dynamisk typinformation....329
26.2. Hur används typkonvertering 329
26.2.1. Praktiskt exempel....331
26.3. Typinformation 331
26.4. Missbruk av typinformation....332
A. C-strängar....334
A.1. Varför C-strängar?....334
A.2. Hur C-strängar fungerar....334
A.2.1. Skapa C-strängar....335
A.2.2. Manipulera enskilda tecken 335
A.3. Tilldelning och jämförelse 336
A.3.1. Tilldelning....336
A.3.2. Jämförelse 337

A.4. Längden av strängar 340
A.5. Kopiering av strängar 341
A.5.1. Färdiga funktioner 343
A.6. Konkatenering av strängar 345
A.7. Andra strängrelaterade funktioner 347
A.7.1. Söka tecken 347
A.7.2. Söka substrängar 349
A.8. Mera information 351
B. Input/output på C:s vis 353
B.1. Introduktion 353
B.1.1. Buffring 353
B.1.2. Headerfiler 353
B.1.3. Konceptet med filer 353
B.2. Utskrift till skärm 354
B.2.1. Utskrift av variabler 355
B.2.2. Utskrift av flyttal 357
B.2.3. Utskrift teckenvis 357
B.3. Utskrift till filer 358
B.3.1. Öppna och stänga filer 359
B.3.2. Skriva data till fil 360
B.3.3. Skriva till filer teckenvis 361
B.3.4. Skriva binär data till filer 361
B.4. Inläsning från tangentbordet 362
B.4.1. Inläsning radvis från tangentbordet 362
B.4.2. Formaterad inläsning 363
B.5. Inläsning från fil 364
B.6. I/O på låg nivå 365
C. Kodlistor 367
C.1. Klassen `Vector` 367
Referenser 372
Index 374

Tabellförteckning

15-1. Skyddsnivåer och arvstyper 204
21-1. Operatorer som kan överlagras 277

Figurförteckning

1-1. C som ett subset av C++ 16
9-1. Schematisk bild över pekare 102
9-2. Schematisk bild över en `int`-vektor 107
9-3. Pekare och minnesadresser 113
9-4. Schematisk bild över pekare 116
11-1. Länkning av bibliotek vid körning 136
13-1. Kompilatorns arbetsskeden 155
13-2. Kompilatorns arbetsskeden vid multipla filer 163
15-1. Klasshierarki 189
15-2. Klasshierarki över figurer 195
16-1. Normal arvshierarki 207
16-2. Multipel arvshierarki 207
16-3. Flera objekt i multipel ärvd klass 209
16-4. Virtuella basklasser 210
21-1. Vektoraddition 280
21-2. Vektorsubtraktion 283
24-1. Uppbyggnaden av en `map` 312
26-1. Arvshierarki 330
A-1. Representation av C-sträng 334

Kapitel 1. Introduktion

Detta kapitel ger en allmän introduktion till C++ och redogör kort för vad detta kompendium skall handla om. Ett enkelt *Hello World*-program genomgås rad för rad.

1.1. Vad är C++

C++ är ett kompilerat, objektorienterat och imperativt språk. C++ är det språk som idag används mest i olika programmeringsprojekt. Många stora system är programmerade med C++, så språket kan användas från de minsta små testprogrammen till komplicerade operativsystem, grafiska gränssnitt och kontorsapplikationer. C++ innehåller funktionalitet för att skriva lågnivåprogram som opererar på hårdvarunivå (t.ex. styrprogram för kringutrustning), men även utmärkta möjligheter för abstraktion på hög nivå. En av fördelarna är alltså att man kan använda C++ för all programmering i ett projekt utan att behöva använda något annat specialiserat språk för sådant som C++ inte klarar av.

För att effektivt kunna använda sig av C++ bör man i de flesta fall lära sig att tänka objektoritenterat. De som kommer från att ha programmerat C en längre tid har ofta svårt att lära sig "tänka om" och istället lära sig abstraktioner. Man kan använda C++ som en form av "C med klasser", men för att fullt kunna utnyttja alla finesser som C++ erbjuder bör man lära sig abstraktioner och ett objektorienterat tänkande.

Namnet *C++* kommer från namnet på programmeringsspråket C. Extensionen ++ är egentligen en *inkrementeringsoperator* (se Avsnitt 3.4) och symboliserar att C++ är C men ändå lite mera än C.

1.1.1. Historia

C++ skapades av Bjarne Stroustup ca. 1979 genom att behov för att objektorienterat språk som var bättre än Simula 67. Bjarne Stroustrup valde att basera sitt nya språk på C, eftersom det var det mest använda vid den tiden. Hade ett annat språk valts som

grund kanske inte C++ idag skulle ha den starka ställning som språket har fått.

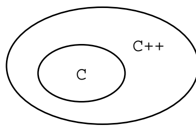
Först kallades C++ för *C with classes*, men ca. 1983 hade man myntat namnet `C++`. Numera är C++ ett standardiserat språk som används av miljoner utvecklare runt om i världen i tusentals olika projekt.

1.1.2. C och C++

C++ bygger som sagt på språket C. C är ett i IT-sammanhang mycket gammalt språk och härstammar från början av 1970-talet, och är skapat av bl.a. Dennis Ritchie och Brian Kernighan.

Tack vare att C++ bygger på C är så gott som allting som är giltig C även giltig C++. C är alltså ett subset av C++.

Figur 1-1. C som ett subset av C++



Det finns några små konstruktioner i C som C++-kompilatorer som följer standarden för C++ inte behöver godkänna. Dessa konstruktioner är sådana som i princip är felaktiga, men som tillåts användas i C-sammanhang. Dessa används ganska sällan, och är ganska enkla ett korrigera ifall man stöter på ett gammalt C-program som en ny C++-kompilator inte godkänner.

1.2. Om detta kompendium

Detta kompendium är skrivet enkom för den första kursen *Programmering i C/C++* hösten 1999. Meningen var att göra ett kort litet kompendium som skulle sammanfatta den viktigaste funktionaliteten i C++. Det hela blev förstås som allting annat "lite större" än vad det var tänkt, och du läser nu resultatet. Personligen är jag ganska nöjd med kompendiet och hoppas att även du som läsare skall ha nytta av mitt arbete. Otvivelaktigt finns det flera viktiga detaljer som fattas och t.o.m. sådant som är direkt fel, men jag fyller i efterhand sådant som fattas och korrigerar om något är fel. Meddela gärna mig om du hittar stavfel, sakfel eller fel i exempelprogrammen.

Ett kapitel med referenser till olika böcker och hemsidor finns sist i detta kompendium. Där finns uppräknade bl.a. de böcker jag själv använt som referenser, samt vissa böcker jag anser vara bra och läsvärda.

1.2.1. Copyright

Detta material är Copyright 1999,2000 Jan Ekholm. Den licens som detta kompendium lyder under är den i Unix-världen mycket använda *GNU General Public License* (se *Referenser*). Denna licens tillåter vem som helst att kopiera detta verk, distribuera och använda det för olika ändamål. Licensen tillåter dock inte någon att sälja detta material eller ens delar av det för egen vinning, och tvingar någon som eventuellt utvidgar materialet att även ge ut sina ändringar för allmän användning och kopiering.

1.2.2. Notation

Detta kompendium använder en ganska enkel notation som följer de flesta andra motsvarande skrifters notation. För text som är viktig, begrepp som introduceras för första gången eller i vissa fall oöversättliga engelska ord används *denna notation*. För ord i den flyande texten som är namn på variabler, klasser, datatyper el.dyl. används `fonten Courier`. På detta sätt är det enkelt att urskilja vad som är vad i texten. För ord som refererar till filnamn används samma font, såsom i t.ex. `Hello.cpp`. Kodexempel är alltid plockade skilt ur den flytande texten, och visas såsom exemplet nedan:

```
cout « "Hello world!" « endl;
```

I HTML-versionen är kodexempel inramade med en annan färg. I vissa fall används *denna notation* inne i kodexempel för att visa att det är frågan om en allmän datatyp eller allmänt variabelnamn. I vissa kapitel finns exempel på kommandon som användaren ger via tangentbordet. Även dessa visas skilt från den normala texten. Det kan se ut t.ex. på följande sätt:

```
% g++ Hello.cpp
%
```

Här används `%` för att symbolisera en *shellprompt* i Unix.

1.3. Ett första program

Vi ska här gå igenom ett första enkelt program skrivet i C++ samt titta en aning på vad det innehåller och var i detta kompendium man kan hitta mera information om just den aspekten av C++.

1.3.1. Hello world!

Det program man traditionsenligt visar som första program är ett som skriver ut texten *Hello world!* på skärmen. Så även i detta kompendium. Programmet kan se ut som exemplet nedan:

```
#include <iostream>

int main () {
  // skriv ut vårt meddelande
  cout « "Hello world!" « endl;
}
```

Vi går rad för rad igenom programmet ovan och tittar på det.

På rad 1 görs en *include*-deklaration (se Kapitel 11 för mera information), som betyder att vi skall använda definitioner i vårt program som finns i filen `iostream`. Denna fil innehåller grundläggande input/output-funktionalitet (se Kapitel 8). Dylika rader börjar med tecknet `#`, och filnamnet skrivs innanför `<>` eller `""`.

Rad 2 är tom. Man kan ha tomma rader i C++-program för att göra dem mera lättläsliga. Det rekommenderas att man gör läsliga program.

Rad 3 definierar en *funktion* som heter `main`. Den tomma parentesen indikerar att funktionen inte behöver några parametrar (se Kapitel 7). Först på raden finns `int`, som betyder att funktionen returnerar ett värde av typen `int`, som är ett positivt eller negativt heltal (se Kapitel 2). Sist på raden finns en `{` som definierar ett *kodblock* (se Avsnitt 1.4.2). Funktionen `main` är alltså ett kodblock som fortsätter ända tills det slutar vid en matchande `}` på rad 6.

Rad 4 är en *kommentar*. En kommentar ignoreras av kompilatorn och finns endast till för att människor skall läsa dem. Se Avsnitt 1.4.4.

Rad 5 sedan gör det egentliga grovjobbet i vårt program. Det skickar *strängen* `Hello world!` till standard utenhet (`cout)`, som i vanliga fall är skärmen. Man använder « för att "styra" texten till `cout`. Efter vårt meddelande skickas ännu ett `endl`, som betyder radbyte. Se Kapitel 8.

Sista raden i programmet avslutar kodblocket som hör till `main`.

1.3.2. Konventioner för namngivning av filer

De flesta språk och filtyper som används i olika sammanhang har en egen *extension*. Extensionen används vanligen för att berätta både för användaren och olika verktyg vilken typ av fil det är frågan om. För t.ex. Modula-2 används extensionen `.mod` medan C använder `.c`. C++ har även egna extensioner. Det finns en del olika extensioner som brukar användas, och man kan i princip använda vilken man själv tycker bäst om. De mest använda är kanske `.cpp`, `.cc`, `.C` och `.c++`. Den sista vållar ibland problem för diverse verktyg och bör undvikas. De flesta verktyg känner igen dessa extensioner som C++-filer. En annan filtyp som används inom C++ (och även C) är en s.k. *definitionsfil*, och de har vanligen extensionen `.h`, `.hh`, `.H` eller `.h++`. Den senaste är inte här heller

speciellt bra att använda. Många kompilatorer kräver inte att filerna är namngivna enligt detta sätt, men det underlättar att hålla ordning på filer om de är systematiskt namngivna.

Det som finns före extensionen är helt valfritt. Man borde dock försöka ge filen ett namn som antingen berättar vad det är för program som finns i filen, eller vilken modul av ett större program det är frågan om. Filnamn som `Fil1.cpp`, `Fil2.cpp` o.s.v. säger ingenting om själva filen, och gör det enbart svårt att senare veta vad filen innehåller.

1.3.3. Kompilera och köra program

Program skrivna i C++ måste *kompileras* innan de kan köras. Men kompilering menas att programmet översätts till en form som den använda datorn direkt kan exekvera. *Tolkade* program däremot måste köras i en tolk som översätter programmet i flykten. Olika kompilatorer kan användas för att kompilera program, men eftersom denna kurs har räkneövningar där vi använder oss av Unix (Linux), så behandlas kompilatorer och verktyg under Unix närmare. De som använder en annan plattform än Unix refereras till dokumentationen som följer med kompilatorn. Alla program i detta kompendium torde dock gå att kompilera med vilken kompilator som helst. Mera information om hur program kompileras finns i Kapitel 13.

Om programmet ovan är sparat i filen `Hello.cpp` kan man under de flesta versioner av Unix kompilera det med ett kommando i stil med:

```
% g++ Hello.cpp
%
```

man får då ett körbart program som heter `a.out` (precis som med kompilatorn för Modula-2). Detta program kan köras med kommandot `./a.out`, och vi får vårt meddelande på skärmen.

```
% ./a.out
Hello world!
%
```

Det var ett första program. Inte så svårt, eller hur? C++ är ganska enkelt, egentligen.

1.4. Grundläggande syntax

Man behöver kunna en del grundläggande syntax innan man kan skriva C++-program. Nedan finns de grundläggande

1.4.1. Separering av satser

En *sats* i C++ kan ses som en instruktion, funktionsanrop, tilldelning eller annat som utför något i C++. Normalt skriver man en sats per rad för att hålla program läsbara, men man kan även kombinera flera satser på samma rad. Satser separeras alltid med ett semikolon (`;`), även om man har en sats per rad. I vissa fall kan man kombinera ihop satser med hjälp av ett kommatecken (`,`), men dylika fall behandlas inte i detta kompendium. Efter tecken som { och } använder man inte semikolon, inte heller efter rader som börjar med `#`. Dylika rader hör egentligen inte till C++, och kommer aldrig att ses av kompilatorn. De är s.k. *preprocessorkommandon* som behandlas före filen egentligen kompileras (se Kapitel 10). Kommentarer behöver inte heller avslutas med semikolon (se Avsnitt 1.4.4. Efter vissa nyckelord behöver skall man inte heller placera ett semikolon, eftersom den efterföljande raden hör ihop med raden med nyckelordet. Hur satser separeras är enkelt, och de exempel som finns i detta kompendium belyser alla olika möjligheter för konstruktion av satser.

1.4.2. Kodblock

I C++ använder man sig av { och } för att begränsa ett kodblock. Ett kodblock är en mängs av noll eller flera satser som hör ihop på något sätt. Det kan t.ex. vara kod som utförs i en slinga eller hör till en funktion. I andra språk används t.ex. `BEGIN` och `END` för samma sak. Vi kommer att stöta på kodblock i varje program vi skriver.

1.4.3. Formatering av kod

Kod kan i C++ formateras nästan hur som helst. Man kan använda tomma rader och

indentering för att göra koden lättare läsbar och mera "luftig", men det är inget tvång. De flesta editorer som används föreditering av C++ stöder automatisk indentering av koden så att man enkelt kan visualisera olika kodblock, men det är ingenting som tvingar en att skriva på det sättet. Vi kunde lika väl ha skrivit det på följande sätt:

```
#include <iostream>
int main(){cout«"Hello world!"«endl;}
```

Vilken man föredrar hoppas jag är självklart. Personligen föredrar jag luftig och indenterad kod som är försedd med rikligt med kommentarer (se Avsnitt 1.4.4), speciellt i inlämningsuppgifter (hint, hint). För kompilatorn är det ingen skillnad hur koden är formatterad, så länge som syntaxen är korrekt. I vissa fall måste man lämna tomrum mellan tecken för att undvika att kompilatorn gör feltolkningar.

Undvik alltså att skriva obskyr kod. Du har lättare att själv läsa din egen kod om några veckor, månader eller år om du behödat dig om att göra den läslig. Du kanske tänker att "inte läser jag min kod någonsin mera" eller "jag förstår den nog". Dylika tankar ledde till att Cobol-programmerare för 20 år sedan tänkte "äh, detta program används inte mera år 2000, jag kan skriva årtalen med två siffror". Vi ser hur det gick. Program har en tendes att överleva sin tilltänkta livslängd. Ofta vill man senare se i sina gamla program hur man löst ett visst problem, hur en algoritm såg ut eller hur syntaxen för någonting fungerar.

1.4.4. Kommentarer

Man kan *kommentera* kod i C++. Med kommentering menas en fri text som skrivs in i programmet och som på något sätt berättar vad programmet gör just på det stället. Kompilatorn ignorerar alla kommentarer i programmet, så ett körbart program blir inte större av att man kommenterar ordentligt. Kommentarer är en form av livförsäkrning då man senare skall försöka förstå vad ett program gör. Normalt tänker man "klart jag vet vad mitt program gör, jag har ju skrivit det". Det kanske stämmer så länge man skriver programmet och allt är i färskt minne, men gör man det efter två månader eller två år?

En kommentar i C++ kan se ut på två olika sätt, beroende på dess längd. Det normala sättet är att använda C++-kommentarer som börjar med två `/`. T.ex. följande innehåller

några komentarer:

```
// ett enkelt exempel-program för att visa hur man kan
// utföra den utskriften av texten "Hello world!"
// till skärmen
int main () {
  // skriv ut vårt meddelande
  cout « "Hello world!" « endl; // vår text
}
```

Kommentarer kan börja var som helst på en rad, men all text *efter* `//` till radens slut tolkas av kompilatorn som en kommentar. Man kan på så vis enkelt kommentera ut kod som man inte vill använda. Om kommentaren fortsätter på flera rader måste ett nytt `//` placeras på varje rad. Om man har långa kommentarer som fortsätter över flera rader kan man använde *block-kommentarer*, som C++ lånat från C. Vi kan skriva om vårt program men C-kommentarer:

```
/* ett enkelt exempel-program för att visa hur man kan
   utföra den utskriften av texten "Hello world!"
   till skärmen */
int main () {
  /* skriv ut vårt meddelande */
  cout « "Hello world!" « endl; /* vår text */
}
```

C-kommentarer har en explicita start- och slutmarkeringar (`/*`) respektive `*/`. Båda måste finnas för att det skall vara en korrekt kommentar. C-kommentarer kan fortsätta över många rader, och används normalt för längre kommentarer, medan C++-kommentarer vanligen används för kortare kommentarer. Vilket sätt man använder spelar ingen roll, båda ignoreras ändå av kompilatorn, huvudsaken är att man kommenterar överhuvudtaget.

Kapitel 2. Variabler och datatyper

Detta kapitel redogör för de olika primitiva datatyperna som finns i C++, samt hur variabler skapas och tilldelas.

2.1. Primitiva datatyper

I C++ finns precis som i de flesta andra programmeringsspråk en mängd olika *datatyper*. Olika datatyper har olika användingsområden och olika aritmetiska operationer fungerar i viss mån olika beroende på datatyp.

2.1.1. Heltal

Den enklaste och vanligaste datatypen är normala *heltal*. Dessa används för diverse beräkningar, iterationer o.s.v. som inte involverar decimaler, endast hela tal. Det finns några olika heltalstyper att välja mellan:

- `int` är grundtypen av heltal. Denna täcker de flesta användningsområden och kan innehålla både positiva och negativa tal. Denna typ är normalt anpassad till datorarkitekturens ordlängd, d.v.s. på en 32-bit system är en `int` även 32 bit stor. Många 64-bit system har dock en 32-bit `int`.
- `long` är normalt en datatyp som är större än en `int`, d.v.s. den kan innehålla större tal. Detta är dock beroende på systemet, men t.ex. flera 64-bit system har en 64-bit stor `long`. En sådan datatyp räcker för att specificera *väldigt* stora tal.
- `short` är kortare än en `int`, ofta endast 16-bit stor. Kan användas då man vet att data som skall finnas i en variabel aldrig är större än att det rymms i ett 16-bit tal.

Alla ovannämnda typer kan innehålla både positiva och negativa tal. Detta halverar dock den teoretiska maximala storleken på de tal datatypen kan representera, eftersom en bit används för att representera tecknet. Ibland kanske man behöver kunna

representera mycket stora tal som garanterat är positiva. Då kan man använda en definition `unsigned` som berättar att datatypen som följer `unsigned` (t.ex. `int`) inte skall använda ett förtecken, utan hela datatypen skall användas för att representera positiva tal. Man får då ett maxvärde som är dubbelt så stort.

2.1.1.1. Talbaser

Man kan i C++ använda olika *talbaser* för att specificera ett tal. I vissa specialtillämpningar är andra talbaser lämpligare än det normala systemet med 10 som grund. Man kan använda *hexadecimala* och *oktala* tal. Den första har basen 16 och den andra har basen 8.

För att specificera ett tal med hexadecimal bas skriver man `0x` eller `0X` (noll-x) före talet. För att använda oktal bas skriver man endast en `0` (noll) före talet. Vi kommer inte i detta kompendium att använda oss av dessa talbaser.

2.1.2. Flyttal

Med *flyttal* i C++ avses tal som har både en heltalsdel och en decimaldel. Flyttal är det som mest liknar normala tal i matematisk mening. Det finns även olika typer av flyttal som har olika precision, precis som heltal ovan. De olika typerna är:

- `float` är den normala versionen på flyttal, precis som `int` för heltal. Denna duger för de flesta tillämningar. Kan innehålla både positiva och negativa heltal. Denna följer vanligen den använda arkitekturens ordstorlek.
- `double` är en flyttalstyp med dubbel precision jämfört med `float`, därav namnet. Den kan användas där en mängd beräkningar med hög precision behövs.
- `long double` är en ännu längre version av `double` för ökad precicion.

Flyttal under C++ lider av vissa speciella problem. Det är omöjligt att representera decimaltal med oändligt antal decimaler m.h.a. några bytes, så många flyttal är avrundningar av de värden de borde innehålla. Om ett stort antal operationer med flyttal

utförs kan detta lilla fel ackumuleras och till sist bli så stort att det är signifikat för slutsvaret. Detta problem kan i viss mån avhjälpas genom att använda större flyttalstyper såsom `double` eller `long double`, men på bekostnad av minne.

2.1.3. Tecken

Tecken representeras i C++ av datatypen `char`. En `char` kan innehålla exakt ett tecken. Denna datatyp är egentligen i grund och botten ett heltal som är en byte långt. Tecken representeras m.h.a. koder på 0 till 255, så dessa ryms exakt i en `char`. Man kan dock göra normal matematisk aritmetik med `char`, samt jämföra tecken som man jämför heltal. Nyare implementationer innehåller en alternativ och längre definition på tecken, nämligen `wchar_t`. Denna är ett mycket nytt tillägg, och används inte ännu i så hög grad. Fördelen med denna är att den kan representera ett mycket stort antal olika tecken, så alla språks speciella tecken kan inkodas i samma datatyp. En normal `char` kan ju samtidigt inte innehålla mera än maximalt 256 skilda tecken.

Även `char` kan ges den extra definitionen `unsigned`, varvid dess omfång är 0 till 255, i normala fall är den -127 till 127.

Teckenkonstanter deklareras genom att innesluta ett tecken mellan två `'`, såsom t.ex. `'a'`, `'9'` eller `'#'`.

2.1.4. Strängar

I C++ finns en datatyp som heter `string` som används för att representera teckensträngar. De som programmerat i C tidigare noterar att detta är ny funktionalitet som uppkommit med de nyaste standarderna av C++. Hur strängar hanteras i C diskuteras i Appendix A. Den nya datatypen är mycket välkommen och gör det lättare speciellt för nybörjare att lära sig C++. En `string` är egentligen mycket mera avancerad än en primitiv datatyp, men vi återkommer till det senare.

En konstant sträng i C++ skrivs innanför citationstecken (`"`). T.ex. är "Hello world!" en giltig sträng. Strängen `""` är en tom sträng. Strängkonstanter kommer att användas extensivt i de flesta av exemplen i fortsättningen för att skriva ut bl.a. informationstext

och resultat. Det finns även några speciella strängkonstanter som används för att utföra speciella funktioner. Dessa är `"\n"` och `"\t"`. Den första används för att skriva ut ett *radbyte*, medan den andra skriver ut ett *tabulatortecken*. Tecknet `\` måste finnas för ett kompilatorn skall förstå att det inte är frågan om ett normalt `n` eller `t`, utan att det är specialtecken. Vi kommer dock inte att använda dessa i våra program, men de flesta program använder dem. Se Kapitel 8 för mera information om vad C++ använder för att byta rad.

2.1.5. Booleans

En annan datatyp som introducerats med i och med C++ är datatypen `bool`. Denna är speciell eftersom den har endast två värden som kan användas, nämligen `true` och `false`. Denna datatyp har många tillämningar, speciellt då olika uttryck skall jämföras, eller då ett visst läge (t.ex. av/på) skall anges. I C emulerade man tidigare datatypen `bool` genom att använda en `int` med värdena 0 eller 1. Många sådana program finns ännu kvar, och endel programmerare föredrar det sättet att representera boolesk funktionalitet. Värdet `true` representeras där av värdet 1 (eller ibland alla värden som inte är 0) och `false` av värdet 0. Program blir dock mycket mera läsbara om man använder `bool` istället för `int`.

2.2. Deklarera variabler

Vi har nu sett vilka olika primitiva datatyper som finns i C++, så det är nu dags att använda dem i program också. Att deklarera variabler i C++ är enkelt. Den allmänna formen för en variabeldeklaration är följande:

```
variabeltyp1 variabel1;
variabeltyp2 variabel2, variabel3;
```

Först på raden kommer alltså namnet på den typ som man vill deklarera, och därefter namnet på den variabel av typen man vill ha. Vill man ha flera variabler av samma typ

sätter man ett kommatecken (`,`) emellan. Glöm inte att avsluta varje deklaration med ett semikolon (`;`). Några exempel på deklarationer:

```
int Resultat;
float Area, Radie;
unsigned int Index1, Index2, Index3;
char Menyval;
```

Varje rad innehåller ju förstås endast variabler av en enda datatyp. Normalt deklareras variabler i C++ i början på funktioner (mera om funktioner i Kapitel 7), men i C++ kan man även deklarera variabler nästan var som helst i en funktion. Om man t.ex. behöver en temporär variabel för något ändamål kan man deklarera den när den behövs. Normalt deklareras variabler dock i början på en funktion för att göra det lättare att läsa koden.

2.2.1. Giltiga variabelnamn

Det finns vissa regler som gäller för hur variabler får namnges i C++. Dessa är desamma som för de flesta andra programmeringsspråk. Giltiga tecken är:

- *stora och små bokstäver* kan användas, precis som man kunde tro. Man bör dock undvika att använda skandinaviska eller andra specialtecken såsom t.ex. *å*, *Ä* eller *ö*. Dessa kanske inte alltid hanteras korrekt av kompilatorn.
- *siffrorna 0 till 9* kan användas, så länge som en variabel inte har en siffra som första bokstav
- *tecknet* _ kan användas för att binda samman ord, men kan även vara första tecken.

Vissa specialvillkor gäller även för benämning av variabler:

- skillnad görs mellan stora och små bokstäver. Variabeln `Data` är således en annan variabel än t.ex. `data`, `DatA` eller `DATA`.
- första teckenet får ej vara en siffra.
- variabelnamnet får ej vara ett reserverat nyckelord i C++ (se kursboken för en lista över reserverade ord).

- inga tomrum tillåts i ett namn. Vill man ha skilda ord kan man använda sig av en *underscore*: _.
- inga begränsingar existerar i C++ på en variabels längd (även om en kompilator kan ha en intern begränsing), och alla tecken i namnet är signifikanta.

Nedan följer några exempel på variabelnamn, både korrekta och felaktiga:

```
int poodle;                     // ok
int PoodlE;                     // ok
Int pipeline                    // ogiltig, måste vara 'int'
int 4ever;                      // ogiltig, börjar på siffra
int TextureMappingParameter3; // ok
int case;                       // ogiltig, 'case' reserverat ord
int Lista-3;                    // ogiltig, innehåller ett '-'
int my_list_1;                  // ok
int _index;                     // ok
```

Ge dina variabler meningsfulla namn. Det är mycket lättare att läsa ett program där variablerna heter t.ex. `area`, `TotalKapital` eller `Index`, jämfört med program där alla variabler heter något i stil med `i`, `j1` eller `tk`.

2.3. Tilldelning av variabler

Nu har vi sett hur man deklarerar datatyper, så nu är det dags att se på hur man kan tilldela dem värden också. C++ är följer även i detta avseende normal praxis. En allmän tilldelning ser ut som nedan:

```
variabel1 = värde1;
variabel2 = variabel3;
datatyp variabel4 = värde2;
```

Man använder sig således av = för tilldelning. En variabel kan tilldelas ett värde av en annan variabel eller ett konstant värde.

```
int Index1, Tal;
```

```
char Tecken;
Index = 10;
Tal = Index;
Tecken = 'C';
```

Man kan även direkt tilldela en variabel som deklareras ett värde, vilket kan vara praktiskt då man deklarerar variabler mitt i en funktion:

```
double TmpSumma = Summa1 + Tal * AnnatTal;
char Default = '1';
```

2.3.1. Typkonvertering

Ibland kan det vara nödvändigt att kunna konvertera mellan olika datatyper. Detta går även att göra i C++, men några undantag och problem. Man kan syntaktiskt korrekt tilldela en variabel av en viss datatyp värdet av en variabel av nästan vilken annan datatyp som helst. Undantaget här är strängar. Det är helt korrekt att tilldela en `double` värdet på en `float`, en `long` värdet på en `int` o.s.v. Om man konverterar heltal till flyttal och tvärtom kan vissa problem förekomma. Om ett heltal tilldelas värdet av ett flyttal kommer decimaldelen att helt enkelt kapas av. Ingen avrundning förekommer. Tvärtom är för det mesta ok. Om flyttalet är väldigt stort och heltalsdelen inte ryms i heltalstypen är resultatet odefinierat.

Om man tilldelar värdet på en variabel till en variabel av en mindre datatyp, t.ex. `int` till `char` kan man få problem. I sådana fall kopieras endast de bitar som ryms i resultatvariablen, och resultatet är numeriskt sällan vad som avses. Det är dock alltid ok att konvertera till en större typ.

C++ sköter om att automatiskt konvertera operander i t.ex. en aritmetisk opertion till en korrekt typ. Om t.ex. två tal adderas och det ena är en `float` och det andra en `int` konverteras även den andra till `float`. Ibland kan man dock vilja göra en manuell explicit typkonvertering. Det gör man genom att skriva en parentes framför variabeln man vill konvertera och i parentesen skriva den önskade typen. Allmänt ser det ut så här:

```
resultattyp variabel1 = (resultattyp)variabel2
```

Några exempel:

```
int Heltal = 10;
float Flyttal = 3.2;
char Tecken = 'w';
int Varde = (int)Tecken;
Flyttal = (float)Heltal;
Heltal = (int)Flyttal;    // förlust av decimaler!
```

I normala fall bör man undvika typkonverteringar för att undvika fel som kan vara svåra att hitta. Planera dina variabler istället på ett bättre sätt så klarar du dig för det mesta utan typkonverteringar!

2.4. Konstanter

I C++ kan man enkelt definiera konstanter med hjälp av nyckelordet `const`. En konstant är en variabel vars värde inte kan ändras under programmets hela körningstid. Konstanter initieras med ett visst värde och därefter kan man använda variabeln men inte ändra denna. Konstanter är praktiska att använda för olika "magiska tal", såsom storlekar på data, olika index eller annat. Allmänt ser en konstant u p följande sätt:

```
const variabeltyp variabelnamn = värde;
```

Några exempel:

```
const float Ranta = 4.12;
const int MaxTal = 100;
const int MinTal = 0;
const string Meddelande = "Hello world!";
```

Notera att man måste ge ett värde åt en konstant då den deklareras. Man kan inte deklarera en konstant och ge värdet på t.ex. nästa rad.

Det rekommenderas varmt att konstanter används alltid då man har ett värde, t.ex. en ränta i ett bankprogram, som behöver ändras ofta. Dessa kan med fördel samlas på ett ställe så att de är lätta att hitta för någon som behöver uppdatera programmet.

Kapitel 3. Aritmetik

Detta kapitel behandlar olika aritmetiska möjligheter i C++. Med detta avses bl.a. hur man kan räkna med variabler, precedens mellan operatorer och binära operatorer.

3.1. Normala aritmetiska operatorer

Man kan exakt som i andra språk använda sig av de normala aritmetiska operatorerna `+`, `-`, `*` och `/` för att manipulera tal och variabler i C++. Detta fungerar precis som i vilket programmeringsspråk som helst. Några exempel:

```
int Summa = 10 + 20;
int Area = Kant1 * Kant2;
float aPris = TotalSumma / Antal;
float Kapital = Kapital + Kapital * Ranta;
```

Man kan även använda `-` för att negera värdet på en variabel, precis som i matematiken. Följande tilldelningar är giltiga:

```
int Negativ = -Positiv;
float Koeff = -10.3 * -Parameter;
```

3.1.1. Jämföra variabler

I C++ används som exemplet ovan visar operatorn `==` för att jämföra variabler med varandra eller med konstanter. Där evalueras uttrycket `Tal == 27` till `true` om `Tal` har värdet 27, i övriga fall får uttrycket värdet `false`. Många andra språk använder endast `=` för att jämföra värden, men så alltså inte i C++.

För att testa om någonting är olika något annat används operatorn `!=`. Denna läses *inte lika med*. T.ex. `Index != 10` evalueras till `true` så länge som `Index` är olika 10.

Storlek kan testas med operatorerna `<`, `>`, `<=` och `>=`. Dessa fungerar precis som sina matematiska motsvarigheter.

3.1.2. Precedens

Med *precedens* avses den ordning enligt vilken komponenterna i ett uttryck evalueras. C++ följer normal matematisk precedens, vilket gör det lätt att konstruera komplicerade matematiska uttryck. Om alla deluttryck är likvärdiga evelueras uttrycken från vänster till höger. Således evalueras följande uttryck såsom kan kunde anta:

```
float NyttKapital = Kapital + Kapital * Ranta;
float TotalArea = Kant1 * Kant2 + Pi * Radie * Radie;
float Procent = Tal / 100 * Koefficient;
```

I det första uttrycket ovan utförs först `Kapital * Ranta` innan det adderas med `Kapital` och sparas i `NyttKapital`. Trots att `+` kom före utfördes ändå `*` först, eftersom den har högre precedens än `+`. `/` och `*` är likvärdiga, likaså `+` och `-`. I det andra fallet är det heller inga problem och summan av de två areorna evalueras sist. Det tredje fallet är dock en aning problematiskt, eftersom både `*` och `/` har samma precedens. Uttrycket evalueras därför från vänster till höger. Om detta *inte* är vad som avses kan precedensen ändras m.h.a. parenteser. Om man vill utföra divisionen först kan uttrycket skrivas om till:

```
float Procent = Tal / (100 * Koefficient);
```

Parenteser kan fritt användas för att ändra precedens, och kan vara bra att använda för att göra ett komplicerat uttryck lättare att läsa. Man kan såldes skriva om det andra uttrycket på följande sätt för att göra det mera läsligt:

```
float TotalArea = (Kant1 * Kant2) + (Pi * Radie * Radie);
```

En komplett precedenstabell med alla operatorer finns i t.ex. kursboken.

3.1.3. Modulus

Med *modulus* avses resten vid heltalsdivision (se Avsnitt 3.2.1). Modulus betecknas med `%`. Några exempel på modulus är:

```
10 % 3 == 1
```

```
6 % 2 == 0
```

Det finns vissa praktiska användingsområden för modulus. Ett exempel är då man vill ha t.ex. entalsdelen från ett valfritt tal:

```
int Heltalsdel = Tal % 10;
```

Andra tillämpning finns även, bl.a. då man manipulerar slumptal och vill ha ett tal som ligger inom ett visst intervall.

3.2. Mera om division

Division fungera inte alltid i C++ såsom man kunde tro. Slutresultatet är beroende på vilka datatyper de tal som divideras har. Det låter ju lite konstigt, men det är både en fördel och en nackdel. Två olika typer av division kan användas: *heltalsdivision* och *flyttalsdivision*.

3.2.1. Heltalsdivision

Om båda operanderna i en division är heltal (`int`, `long` o.s.v.) kommer slutresultatet även att vara ett heltal. Detta innebär att en eventuell decimaldel kommer att totalt ignoreras. Således är följande uttryck sanna:

```
9 / 4 == 2
(100 / 30) * 30 != 100
```

Det verkar konstigt, men det är så det fungerar. För att råda bot på detta problem bör man använda sig av *flyttalsdivision*.

3.2.2. Flyttalsdivision

Flyttalsdivision fungerar såsom division matematiskt normalt fungerar. Där fås alla decimaler till resultatet. Det enda villkoret för att flyttalsdivision skall användas istället för heltalsdivision är att minst ena av operatorerna måste vara ett flyttal (t.ex. `float`, `double`). Exemplen nedan använder sig av flyttalsdivision:

```
float Resultat = 9.0 / 4.0;
float Resultat = 9 / 4.0;
float Resultat = 9.0 / 4;
float Resultat = (float)9 / 4;
```

I dessa exempel är minst ena av operanderna ett flyttal. I det sista exemplet är `9` inget flyttal, men där används en typomvandling för att omvandla denna till ett flyttal. En konstant är alltid ett flyttal om den har en decimaldel, såsom i `4.0`, även om talet matematiskt egentligen är ett heltal.

3.2.3. Precisionsproblem

Flyttal i C++ har inte oändlig precision, så ibland kan man få speciella resultat efter att man dividerar med flyttal (och multiplicerar, adderar och subtraherar) beroende på precisionsproblem. Alltid är t.ex. `10.0 / 5.0` inte exakt `2.0`, utan det *kan* vara t.ex. `2.0000001`. Om dylika fel uppkommer och ackumuleras kan man med tiden få ett fel som är signifikant. Man kan undvika detta problem till en viss del egenom att använda sig av flyttalstypen `double` istället för `float`, eftersom den har dubbel precision jämfört med `float`, och således lider i en mindre grad av ackumulerande fel.

Det går således inte alltid att direkt jämföra ett flyttal med en konstant, eftersom flyttalet kan vara minimalt större eller mindre än konstanten man jämför med. Detta kan vara bra att komma ihåg om man hanterar flyttal och vill jämföra dem med varandra.

3.2.4. Division med `0`

Division med 0 är något som man normalt bör undvika. Då man utför divisioner bör nämnarens värde kontrolleras för att undvika division med 0. Beroende på systemet kan

detta leda till en omedelbar terminering av programmet, eller så att divisionen resulterar i ett speciellt värde som resulterar ett ogilitigt eller oändligt värde. I vissa fall kan en *exception* kastas (se Kapitel 20).

3.3. Binär aritmetik

Med binär aritmetik avses att de involverade talen behandlas som en serie `1`:or och `0`:or, och manipuleras *bitvis*. En *bit* är en enskild `1`:a eller `0`:a. I vissa fall är det praktiskt att kunna behandla tal på den lägsta möjliga nivån, d.v.s. bitnivån. Speciellt då man programmerar hårdvarunära applikationer. För mera information om bitvis representation av tal se t.ex. introduktionsmaterial till informationsbehandling eller material om digitalelektronik.

3.3.1. Shiftoperatorerna « och »

Shiftoperatorerna « och » används för att shifta den binära representationen av ett tal till vänster (talet * 2 upphöjt till högra argumentet) och till höger (integerdivision med 2 upphöjt till högra argumentet). Några exempel visar vad det är frågan om:

```
10 « 2 == 40
15 « 3 == 90
10 » 1 == 5
10 » 2 == 2
```

Att shifta vänsterut med `1` kan ses som ett enkelt sätt att fördubbla ett tals värde.

Varning

Blanda inte ihop shiftoperatorerna « och » med I/O-operatorerna med samma namn som C++ använder för att läsa ock skriva streams! Om du använder shiftoperatorer i samband med I/O bör du för att undvika fel använda parenteser för att försäkra dig om att shiftningarna utförs före eventuell I/O. För mera info om C++ I/O-funktioner se Kapitel 8.

3.3.2. Bitvisa operatorerna |, &, ^ och ~.

Dessa fyra operatorer kan användas för att manipulera enskilda bitar i ett tal. Detta kan behövas om t.ex. något standard bibliotek råkar använda sig av enskilda bitar i någon parameter för att representera någon viss funktionalitet, eller när man programmerar hårdvarunära och behöver ändra en viss bit i ett register.

För att göra en viss bit i ett tal till 1 kan man använda sig av operatorn |. Detta är en vanlig binär *or*. För att ändra läge på en viss bit kan man använda sig av operatorn ^, som fungerar som en normal *exclusive or*. Att "maska" ut vissa bitar ur ett tal kan man göra med &, som är en normal binär *and*. Sist finns operatorn ~ som är en binär *not*, och inverterar ett tals binära representation. Några exempel visar hur dessa operatorer fungerar:

```
// sätta en viss bit eller flera bitar
int Resultat = Tal | bitmask;
int Resultat = Tal | 64;

// ändra värde på en viss bit
int Resultat = Tal ^ bitmask;
int Resultat = Tal ^ 32;

// maska ut vissa tal, t.ex. de lägsta 8 bitarna
int Resultat = Tal & bitmask;
int Resultat = Tal & 255;
```

```
// nollställa en viss bit (negering först)
int Resultat = Tal & ~bitmask;
int Resultat = Tal & ~8;
```

Det sista exemplet kan behöva lite förklaring. För att nollställa en viss bit inverteras först den bitmask som man vill nollställa. Därefter görs en *and* med originaltalet. Detta har som resultat att endast den biten som skall nollställas är 0 i masken, och enligt hur *and* fungerar blir resultatet då det ursprungliga talet med den ena biten (eller flera bitar, om så önskas) nollställd.

3.4. Inkrementerings- och dekrementeringsoperatorerna ++ och −−

C++ baseras sig som bekant på C, och C har alltid varit alla lata programmerares favoritspråk. Det finns en mängd förkortningar och olika sätt att skriva saker i C, och då även i C++. En operation som man märkte att ofta förekom i tidiga C-program var en inkrementering eller dekrementering av ett tal med 1, t.ex:

```
int Index, Ar;
Index = Index - 1;
 Ar = Ar + 1;
```

För att göra det hela kortare att skriva uppfanns operatorerna ++ och >−−. Dessa används för att inkrementera eller dekrementera sitt argument. De ovanstående exemplen skulle med dessa operatorer kunna bli:

```
int Index, Ar;
Index>−−;
Ar++;
```

En hel del kortare eller hur? För att göra det mera intressant kan man placera operatorerna antingen *framför* eller *bakom* operanden. Om operatorerna är framför kallas med dem för *prefixa* och är de bakom är de *postfixa*. Beroende på placeringen

påverkas *när* inkrementeringen/dekrementeringen utförs. En prefixoperator evalueras *före* operanden används i andra uttryck, medan en postfixoperator evalueras *efter* att operanden använts i andra uttryck. Således är dessa två uttryck inte ekvivalenta:

```
int Tal1 = 10;
int Tal2 = 10;
cout « "Tal1++ = " « Tal1++ « endl;  // ger svaret 10
cout « "++Tal2 = " « ++Tal2 « endl;  // ger svaret 11
```

Det första fallet skriver ut `Tal1` innan det inkrementeras och ger svaret 10, medan det senare exemplet inkrementerar före `Tal2` används, och ger således svaret 11. Man bör vara medveten om denna fundamentala skillnad då man använder dessa operatorer. De är dock mycket praktiska att ha då man t.ex. itererar (se Kapitel 5) och vill öka/minska en indexvariabel med 1.

3.5. Sammansatta tilldelningar

Som tidigare nämnts är C++ ett språk för lata programmerare som inte vill skriva mycket. Ovan visades hur man kan använda prefix- och postfixinkrementering/dekrementering för att spara lite utrymme och göra program kortare. Det hela kan generaliseras till vilka aritmetiska operatorer som helst som ändrar ett tal och sparar resultatet i samma variabel. Om vi har det generella fallet:

```
Tal1 = Tal1 operator Tal2;
```

kan man generalisera detta genom att totalt lämna bort den andra instansen av `Tal1` och dra in *operator* så att den är fast i `=`, men på vänster sida. Några exempel på det traditionella sättet att skriva operationerna:

```
int Tal;
Tal = Tal + 10;
Tal = Tal * 5;
Tal = Tal % 2;
Tal = Tal | 32;
```

Kan på det kompakta sättet skrivas:

```
int Tal;
Tal += 10;
Tal *= 5;
Tal %= 2;
Tal |= 32;
```

Det hela blir nog kortare, men förlorar en del av läsbarheten. I övrigt ger de exakt samma resultat som den längre versionen av samma operation. Vilken man väljer beror närmast på vilken som känns mera naturlig och vilken man är säkrare på.

Kapitel 4. Enkel input/output

Detta kapitel går igenom mycket elementär *input/output* i C++. Orsaken till detta är att de kommande kapitlen behöver enkel I/O för exempelprogrammen, men det egentliga I/O-kapitlet kommer först senare, och det i sin tur bygger på de kapitel som kommer före. För att råda bot på denna "ägget och hönan"-situation tas här upp input från tangentbord och output till skärm. Mera information finns sedan i Kapitel 8, där I/O till skärm och från tangentbord behandlas i mera detalj.

4.1. Utskrift till skärmen

Utskrifter till ksärmen sköts via entiteten `cout`. Denna representerar utskrifter till skärmen i C++. För att skriva ut data på skärmen måste man "styra" informationen till `cout` med hjälp av «. Allmänt ser det ut på följande sätt:

```
cout « konstant;
cout « variabel;
cout « variabel1 « variabel2 « ... « variabelN;
```

Allting som styrs till `cout` skrivs ut. `cout` är ganska intelligent så man kan utan problem styra ut alla datatyper vi redan känner till, t.ex. `string`, `int` och `float`, utan att bekymra oss om hur `cout` sköter uppgiften. Flera utskrifter kan *konkaterneras* genom att placera flera « efter varandra med variabler eller konstanter emellan. Några exempel på utskrifter:

```
int Tal;
float Tid;
string Svar;
cout « "Hello world!";
cout « Tal;
cout « "Svaret är: " « Svar;
cout « "Ping-tiden = " « Tid « " millisekunder";
```

Man kan förstås bryta rader som blir för långa och placera resten av utskriften på en annan rad.

4.2. Inmatning från tangentbordet

Inmatning i C++ sköts med en motsvarande entitet som heter `cin`. Med denna läser man in data från tangentbordet i variabler. Där vänder man om "styrtecknen" så att de blir », och således "pekar" in mot variablerna. Allmänt ser det ut på följande sätt:

```
cin » variabel;
cin » variabel1 » variabel2 » ... » variabelN;
```

Beroende på datatypen på den variabel som följer efter » förväntar sig `cin` att användaren skall skriva in ett sådant värde på tangentbordet och trycka *enter*. Läses flera variabler på samma gång så hanterar `cin` dem från vänster till höger. Att läsa in ett enkelt tal:

```
int Tal;
cout « "Ge in ett tal: ";
cin » Tal;
```

4.3. Headerfilen `iostream`

Alla program som vil använda sig av `cout` och `cin` bör inkludera `headerfilen iostream` enlit följande:

```
#include <iostream>
```

Den innehåller diverse nödvändiga definitioner.

Kapitel 5. Iteration

Detta kapitel behandlar de olika iterationsmöjligheterna i C++. Tre olika typer av slingor kan användas, alla med olika användingsområden. Med iteration avses en eller flera satser som upprepas noll eller flera gånger beroende på ett visst villkor.

5.1. `while`-slingor

Den enklaste av alla slingor i C++ är den såkallade `while`-slingan. Den används för att upprepa ett antal satser så länge som ett villkor håller. Den allmänna formen ser ut så här:

```
while ( villkor )
    sats;
```

`while`-slingan upprepar `sats` så länge som *villkor* evalueras till `true`. Om villkoret aldrig är sant utförs slingan aldrig, utan exekveringen fortsätter direkt på raden efter. Villkoret evalueras efter varje exekvering av slingan. Exemplet nedan skriver ut talen från 1 till 10 på skärmen:

```
// skriver talen 1 till 10
int Index = 1;
while ( Index++ <= 10 )
    cout « "Tal: " « Index « "\n";
```

Notera att värdet på `Index` ökas *efter* att `Index` använts, d.v.s. första iterationen har den värdet 1, och då iterationen är slut har den värdet 11. Vill man ha flera än en sats i slingans kropp bör man använda ett block av satser:

```
while ( villkor ) {
    sats1;
    sats2;
    sats3;
}
```

I detta fall utförs alla tre satser sekventiellt. Villkoret i `while`-slingan kan vara vad som helst som evaluerar till något som kan tolkas som `true` eller `false`. Det är rekommendabelt att alltid kapsla in `while`-slingans sats(er) i block, även omd et endast är frågan om en enda sats. Det är lätt hänt att man senare vill utöka slingans innehåll, och om man då inte har satserna inom ett block händer det enkelt att man får något i stil med detta:

```
// beräknar en summa av talen 1 till 10
int Index = 1;
int Summa = 0;
while ( Index <= 10 )
    Summa = Summa + Index;
    Index++;
cout « "Summan är: " « Summa « "\n";
```

Programmet ser helt korrekt ut med en snabb titt, men ser man närmare på det märker man snabbt att endast raden `Summa = Summa + Index;` tillhör `while`-slingans kropp, och således är den enda rad som utförs. Raden `Index++` verkar enligt indenteringen att tillhöra slingan, men gör det inte. Resultatet blir en evig slinga och inget svar. Det korrekta förfarandet är att använda ett kodblock för slingans kropp:

```
// beräknar en summa av talen 1 till 10
int Index = 1;
int Summa = 0;
while ( Index <= 10 ) {
    Summa = Summa + Index;
    Index++;
}
cout « "Summan är: " « Summa « "\n";
```

Vi kommer att se på alternativa sätt att skriva slinga ovan senare i detta kapitel.

5.1.1. Eviga slingor

En *evig slinga* är en slinga som aldrig avslutas. Dylika kan vara praktiska då ett program skall köra en viss slinga tills det terminerar. Eftersom villkoret i en `while`-slinga är flexibelt kan en evig slinga skrivas t.ex. på följande sätt:

```
while ( true ) {
    cout « "Hello world!\n";
}
```

I exemplet ovan kan slingans vilkor aldrig evalueras till `false`. Vi kunde även skrivit vilkoret som `1` eller ett annat tal som inte är 0, eftersom de evalueras till `true`.

5.1.2. Nästlade slingor

Ibland kan det vara praktiskt att kunna nästla slingor. Detta kan förstås göras även med de slingor som finns i C++. Man kan ha flera slingor innanför varandra, med det enda kravet att de måste vara helt innanför varandra, de kan inte överlappa på något sätt. Olika typer av slingor kan förstås även nästlas. Exemplet nedan skriver ut talen 1 till 100 i en fyrkant, med 10 tal per rad:

```
int Rad = 0;
int Kolumn = 1;
while ( Rad < 100 ) {
    while ( Kolumn < 10 ) {
        // skriv ut ett tal och inkrementera kolumnen
        cout « Rad + Kolumn;
        Kolumn++;
   }

   // en rad full, gå till nästa rad
   cout « "\n";
   Rad = Rad + 10;
}
```

Vi kommer senare i detta kapitel att se ett elgantare sätt att utföra samma sak, men med hjälp av `for`-slingor (se Avsnitt 5.3).

5.2. `do-while`-slingor

Denna slinga är i princip ekvivalent med en normal `while`-slinga, med den enda skillnaden att denna utför villkorstestet *efter* att slingans kropp utförts en gång. Denna slinga är således garanterad att utföras minst en gång, även om villkoret aldrig evalueras till `true`. Allmänna formen för en `do-while`-slinga är :

```
do
    sats
while ( villkor );
```

Även här rekommenderas att kodblock används för att bättre markera slingans kropp:

```
do {
    sats
}
while ( villkor );
```

Var blockmarkeringarna placeras är förstås upp till var och en att själv avgöra och placera enligt eget tycke och smak. `do-while`-slingor används relativt sällan jämfört med vanliga `while`-slingor eller `for`-slingor.

5.3. `for`-slingor

Denna typ av slinga är den mest flexibla och troligtvis den mest använda av de tre typerna av slingor i C++. Den tillåter relativt avancerad iteration med hjälp av en enda slinga. Den allmänna formen för en `for`-slinga är:

```
for ( initiering; villkor; inkrementering )
    sats
```

Slingan har tre olika moment som utförs vid olika tidpunkt då slingan utförs:

1. *initiering* som utförs en enda gång då slingan skall starta. Här kan en variabel t.ex. initieras till ett startvärde.

2. *villkor* som avgör när slingan skall sluta. Detta villkor fungerar på samma sätt som `while`-slingans, och skall evalueras till `true` eller `false`. Slingan utförs så länge villkoret evalueras till `true`. Villkoret kontrolleras före varje exekvering av slingans kropp.
3. *inkrementering* används för att ändra värdet på t.ex. en variabel som kontrollerar slingan. Denna utförs *efter* varje exekvering av slingans kropp.

`for`-slingan är mycket kraftfull och flexibel tack vare de tre olika deluttrycken som kan hanteras totalt oberoende av varandra. Ett exempel på en slinga som summerar talen 1 till 10 kan se ut som nedan:

```
// beräknar en summa av talen 1 till 10
int Index;
int Summa = 0;
for ( Index = 1; Index <= 10; Index++ ) {
    Summa = Summa + Index;
}
cout « "Summan är: " « Summa « "\n";
```

I slingans *initieringsdel* som utförs en gång initieras `Index` till 0. Denna del utförs endast en gång. Därefter utförs *villkorsdelen* som ju kommer att evalueras till `true` så länge som `Index` <= 10. Slingans kropp utförs därefter och summan ökas. Därefter utförs *inkrementeringsdelen* som ökar `Index` med 1, varpå *villkorsdelen* kollas igen och slingan fortsätter. `for`-slingor kan verka kryptiska till en början, men efterhand lär man sig uppskatta dess kraftfulla syntax som ger möjligheter till diverse flexibla och kompakta kontruktioner. Ett exempel som gavs ovan som skriver ut talen 1 till 100 i en fyrkant kan med hjälp av `for`-slingor se ut som nedan:

```
for ( int Rad = 0; Rad < 100; Rad += 10 ) {
    for ( int Kolumn = 0; Kolumn < 10; Kolumn++ ) {
        // skriv ut ett tal
        cout « Rad + Kolumn « " ";
   }
   // en rad full, gå till nästa rad
   cout « "\n";
}
```

Detta exempel visar hur den yttre slingan använder sig av ett speciellt inkrementeringssteg där raden ökas med 10 per iteration. Exemplet använder sig även av möjligheten att i C++ deklarera variabler där de behövs. I exemplet är `Rad` och `Kolumn` deklarerade i slingornas initieringsdelar. Deras *omfång* (scope) är således endast inom slingorna. Detta är ett praktiskt sätt att deklarera temporära iterationsvariabler då de behövs.

Man behöver inte fylla i alla de tre delarna av en `for`-slinga. Om det t.ex. inte finns någon naturlig initieringssats kan man lämna bort denna. Lämnar man bort alla delar får man en *evig slinga*:

```
// en evig slinga
for ( ; ; ) {
  cout « "Hello, forever!" « endl;
}
```

Det är dock normalt endast i specialfall man vill lämna någon del tom.

5.4. Avbryta och fortsätta slingor

5.4.1. Avbrytning med break

I vissa fall kan det vara nödvändigt att i en slingas kropp kunna antingen avsluta hela slingan eller hoppa till slingans start. Det första kan göras med en `break`-sats. Denna avslutar den innersta slingen och fortsätter på raden omedelbart efter slingan. Allmänt ser det ut t.ex.:

```
while ( villkor ) {
    sats1;
    break;
    sats2;
}
sats3;
```

I detta fall kommer `break` att omedelbart avsluta slingan och hoppa till raden direkt efter slingan, d.v.s. raden där `sats3` finns. `sats2` kommer således aldrig att utföras! Ett användingsområde för slingor i iterationssammanhang är för att avsluta en evig slinga:

```
// en evig slinga med hjälp av en for-slinga
for ( ; ; ) {
    // gör något
    ...

    // ett villkor A säger att vi skall avsluta slingan
    if ( A ) {
        break;
    }
}
```

I de flesta fall kan man dock undvika att använda en dylik konstruktion (som ju kan vara svårläslig i vissa fall) genom att göra en normal `while`-slinga med villkoret A som slingans villkor. Andra användingsområden för `break` finns i kapitlet om flödeskontroll (se Kapitel 6).

5.4.2. Fortsättning med `continue`

I motsats till att avsluta en slinga tvärt kanske man vill kunna göra något som säger att "denna iteration är nu klar, testa villkoret på nytt och fortsätt därifrån". Detta görs med hjälp av satsen `continue`. Den allmänna formen för `continue` är:

```
while ( villkor ) {
    sats1;
    continue;
    sats2;
}
sats3;
```

I detta fall kommer `continue` att åstadkomma att exekveringen av slingan fortsätter från *villkor*. Resten av slingan hoppas således helt över, och `sats2` kommer även i detta fall aldrig att utföras! Slingan avslutas dock inte, utan endast *villkor* kan avgöra detta.

Konstruktioner med `continue` kan vara praktiska om man har djupt nästlade villkorssatser och vill kunna "starta om" en slinga som innehåller dessa. Då en `for`-slinga används åstadkommer `continue` att exekveringen fortsätter från slingans *inkrementeringsdel*, medan `while`- eller `do-while`-slingor fortsätter exekveringen från slingans *villkorsdel*. Då man använder `while`-slingor bör man således se upp så man inte i misstag hoppar över den delen av slingan som t.ex. inkrementerar en variabel eftersom det skulle åstadkomma en evig slinga.

Kapitel 6. Flödeskontroll

Detta kapitel tittar närmare på de olika möjligheterna till *flödeskontroll* som finns i C++. Med flödeskontroll avses att ett program kan utföra olika satser på bas av ett villkors värde.

6.1. `if`-satser

Den enklaste former av flödeskontroll i C++ är en `if`-sats. Denna kontrollerar att villkor och om detta evalueras till `true` utförs `if`-satsens kropp, i annat fall ignoreras kroppen. Den allmänna formen ser ut så här:

```
if ( villkor )
    sats;
```

Även `if`-satser bör använda sig av kodblock för att undvika misstag. Ett exempel som kontrollerar om ett tal har ett visst värde kan se ut så här:

```
// ge en variabel Tal ett värde
int Tal;
cin » Tal;
if ( Tal == 27 ) {
    // talet var 27
    cout « "Talet var 27!\n";
}
```

I C++ används inte nyckelordet `then` eller dylikt såsom i t.ex. Modula-2. Exemplet ovan sktiver ut *Talet var 27!* om `Tal` har värdet 27.

`if`-satser är den mest använda konstruktionen för att styra ett programs flöde i C++. Så gott som alla program använder sig av `if`-satser. Man bör dock vara noggrann med att man faktiskt skriver `==` då man avser att göra en jämförelse. Vad händer om man istället hade skrivit:

```
if ( Tal = 27 ) {
```

```
    ....
}
```

Beroende på kompilatorn som används är detta helt korrekt C++ och inte ens en varning ges. Här har ett = fallit bort. Kvar blir då en tilldelning! Det som händer är att `Tal` tilldelas värdet 27 och `Tal` evalueras då till `true` ftersom det inte är 0, och `if`-satsens kropp kommer att utföras. Kanske inte direkt önskat resultat. Detta är ett vanligt fel, och kan vara svårt att hitta.

6.1.1. Gruppering av kod i block

Då man hanterar `if`-satser är det speciellt viktigt att tänka på att använda block runt `if`-satsen kropp. Speciellt om man avser att ha fler än en enda sats som kropp. Exemplet nedan fungerar inte som man kunde tro

```
// läs ett menyval från tangentbordet
char Menyval;
cin » Menyval;
if ( Menyval == 's' )
    // spara någonting till fil
    cout « "Sparar data till fil." « endl;
    spara ();
```

Enligt indenteringen ovan så skulle funktionen `spara()` (mera om funktioner i Kapitel 7) utföras endast då menyvalet är ett *s*, men så är inte fallet. Endast raden med `cout` tillhör `if`-satsens kropp, raden med `spara()` kommer efter hela `if`-satsen och utförs således alltid. Kanske inte önskat beteende. Problemet åtgärdas genom att kapsla in kroppen mellan { och }:

```
// läs ett menyval från tangentbordet
char Menyval;
cin » Menyval;
if ( Menyval == 's' ) {
    // spara någonting till fil
    cout « "Sparar data till fil." « endl;
    spara ();
}
```

Att använda block är något som är bra att göra till en vana. Koden blir minimalt längre, men antalet onödiga logiska fel minskar.

6.1.2. Nästlade `if`-satser

Det är fullt möjligt att nästla `if`-satser innanför varandra om det skulle behövas, precis som i andra programmeringsspråk. Detta kan se ut som nedan där ett tal först kontrolleras om det är jämnt, och sedan om det är delbart med 4 eller 6.

```
// ge en variabel Tal ett värde
int Tal;
cin » Tal;
if ( Tal % 2 == 0 ) {
    // talet var jämnt, d.v.s. delbart med 2, är det delbart med 4?
    if ( Tal % 4 == 0 ) {
       // talet var delbart med 4
      cout « "Talet " « Tal « " är delbart med 4\n";
    }

    // är det delbart med 6?
    if ( Tal % 6 == 0 ) {
       // talet var delbart med 6
      cout « "Talet " « Tal « " är delbart med 6\n";
    }
}
```

6.2. `if-else`-satser

Ibland kan det vara praktiskt att kunna utföra en del av ett program om en `if`-sats evaluerats till `false`, d.v.s. om den har inte utförts. Det är förstås möjligt att skriva en andra `if`-sats som kontrollerar det negerade uttrycket jämfört med den första, men det hela kan göras lättare genom att skriva en `else`-del till `if`-satsen i fråga. Denna `else`-del utförs endast om `if`-satsen evaluerades till `false`. Den allmänna formen är:

```
if ( villkor )
    sats1;
else
    sats2;
```

I en `if`-sats garanteras att antingen `if`-delen *eller* `else`-delen utförs, aldrig båda eller ingendera. Ett exempel med talet 27 som fanns tidigare kunde bli:

```
// ge en variabel Tal ett värde
int Tal;
cin » Tal;
if ( Tal == 27 ) {
    // talet var 27
    cout « "Talet var 27!\n";
}
else {
    // talet var inte 27
    cout « "Talet var inte 27!\n";
}
```

Det går även att kombinera ihop flera `if`-satser med varandra för att åstadkomma en lång rad olika fall. Det hela belyses bäst med ett exempel:

```
// ge en variabel Tal ett värde
int Tall;
cin » Tal;
if ( Tal < 20 ) {
    // talet är mindre än 20
    cout « "Talet är mindre än 20!\n";
}
else if ( Tal < 25 ) {
    // talet är mindre än 25, man större eller lika med 20
    cout « "Talet är större eller lika med 20, men mindre än 25!\n";
}
else {
    // talet är större eller lika med 25
    cout « "Talet är större eller lika med 25!\n";
}
```

Exemplet ovan är kanske inte så speciellt realistiskt, men illustrerar hur en ny `if`-sats inkluderats i `else`-delen av en annan sats. Den sista `else`-delen utförs endast om inget av de ovanstående villkoren uppfyllts.

6.3. Logiska operatorer

Den allmänna formen för en `if`-sats är ju:

```
if ( villkor )
    sats;
```

Delen *villkor* har i exemplen ovan varit ett enda uttryck, men C++ tillåter att flera uttryck grupperas samman med hjälp av logiska operatorer. På detta sätt kan man uttrycka förhållanden som *och*, *eller* och *inte*. Operatorerna är i C++ `&&`, `||` respektive `!`.

> **Varning**
>
> Blanda inte ihop de logiska operatorerna `&&` och `||` med de binära operatorerna `&` och `|`. I många fall är det helt korrekt C++ att använda de binära operatorerna i t.ex. `if`-satser, och kompilatorn kommer att godkänna det, även om det inte är vad som avses

Ett kort exempel på hur operatorn `&&` kan användas för att kombinera två uttryck:

```
float Rot;
cin » Rot;
// är Rot inom ett litet intervall från Pi?
if ( Rot > 3.14158 && Rot < 3.14160 ) {
    // talet är väldigt nära Pi
    cout « "Talet " « Tal « " är nära talet Pi." « endl;
}
```

Här kommer `if`-satsens kropp att utföras om och endast om `Rot` uppfyller båda kraven, det räcker inte med endast det ena. För att kolla om ett uttryck *eller* ett annar gäller

illustreras av följande exempel. Här visas även hur flera än två uttryck kan kombinera tillsammans.

```
char Menyval;
cin » Menyval;
// skall foo utföras?
if ( Menyval == ’s’ || Menyval == ’S’ || Menyval == ’1’ ) {
    // jep, utför foo
    foo ();
}
```

Operatorn `!` används för att negera en utsaga. Ett uttryck för *olika med* kan således skrivas:

```
if ( ! Ar == 1972 ) {
    ...
}
```

Ovannämnda exempel kunde mera lättläst skrivas `Ar != 1972`. När man konstruerar logiska uttryck bör man tänka på operatorernas `precedens` (se Avsnitt 3.1.2) för att undvika uttryck som evalueras på fel sätt. Är man inte säker på precedensen kan man kombinera uttryck med parenteser (se Avsnitt 6.3.2).

6.3.1. Lazy evaluation

C++ använder sig av en finess som kallas *lazy evaluation*. Detta innebär att då sanningsvärdet för en utsaga beräknas evalueras uttryck från vänster till höger ända tills utsagans värde är klart. Om utsagans värde klarnar för hela uttrycket evaluerats ignoreras resten. I en allmän form med operatorn `&&` kan det se ut:

```
if ( A && B && C && D ) {
  ...
}
```

Antag att `A, B, C` och `D` är uttryck av något slag. Eftersom `&&` sammanbinder dem alla måste alla uttryck evalueras till `true` för att `if`-satsen skall utföras. Om t.ex. `B` evalueras till `false` kommer ju hela uttrycket att få värdet `false`, och det är ingen ide

att evaluera `C` och `D`. I ett ekvivalent exempel, men med `&&` ersatt med `||` kommer evalueringen ett stanna så fort ett av uttrycken evalueras till `true`, eftersom hela uttrycket då måste evalueras till `true`.

Ett bra exempel på hur lazy evaluation ofta används är:

```
if ( X != 0 && 4/X < 3 ) {
  ...
}
```

Här kan vi ju i misstag dividera med 0 om inte kontrollen `X != 0` funnits. Om `X` har värdet 0 kommer hela uttrycket i vilket fall som helst att få värdet `false`, så `4/X < 3` evalueras aldrig.

6.3.2. Kombinera uttryck

För att kombinera uttryck tillsammans kan man använda sig av normala parenteser, precis som i logiken. Om man t.ex. vill att uttycket A eller B skall gälla, *och* uttrycken C eller D, kan man skriva det på följande sätt:

```
if ( ( A || B ) && ( C || D ) ) {
  ...
}
```

Man bör komma ihåg att operatorn `&&` har högre prcedens än `||`, så ovanstående hade utan parenteser egentligen tolkats på följande sätt:

```
if ( A || (B && C) || D ) {
  ...
}
```

Alltså inte exakt vad som avsågs. Parenteser evalueras först, börjande från den innersta parentesen, precis enligt normala precedensregler. Det kan vara värt att kontrollera sina logiska uttryck om man har ett program som beter sig konstigt.

6.4. `switch`-uttryck

C++ har en konstruktion som är mycket praktisk då man vill välja en av flera olika alternativ. Om vi t.ex. har en meny där användaren kan välja alternativen 1, 2, 3 eller 4 kan vi implemntera det hela som en som `if`-konstruktion, men det är ganska klumpigt och väldigt fult. Istället kan man använda `switch`-uttryck. Ett exempel illustrerar vad det är frågan om:

```
// läs ett menyval från tangentbordet
char Menyval;
cin » Menyval;
switch ( Menyval ) {
  case ’1’ : cout « "1 valdes" « endl; break;
  case ’2’ : cout « "2 valdes" « endl; break;
  case ’3’ : cout « "3 valdes" « endl; break;
  case ’4’ : cout « "4 valdes" « endl; break;
}
```

Ovan har vi först en sats med nyckelordet `switch` som börjar `switch`-blocket. Denna tar som argument ett uttryck som skall evelueras till ett *heltal*. Observera att det skall vara ett heltal, inte ett boolskt uttryck som i `if`-satser. På bas av detta heltal kontrolleras varje `case`-sats för att se ifall värdet är samma. Om så är utförs `case`-satsens kropp. Så fortsätter evalueringen nedåt i `switch`-konstruktionen ända till slutet nåtts utan ’träffar’, eller en eller flera `case`-satser utförts. Observera att syntaxen där ett tecken sätts innanför ” ju ger tecknets ASCII-värde, d.v.s. ett tal. Den allmänna formen för en `case`-konstruktion är:

```
switch ( evaluering ) {
  case värde1 : sats1;
  case värde2 : sats2;
  ...
  case värdeN : satsN;
  default: sats;
}
```

Värdena *värde1* till *värdeN* måste vara distinkta, d.v.s. inga kopior tillåts förekomma. Den sista satsen `default` utförs normalt om ingen av de ovanstående `case`-satserna

utfördes, och fungerar som en form av uppsamlare för alla värden som inte testats. Man kan se på `default` som den sista `else`-delen i en konstruktion med många `if`-satser.

Koden i en `case`-kropp är execpetionell på det viset att man inte där behöver gruppera koden i block. Satser utförs nedåt ända tills slutet av `switch`-konstruktionen nåtts, eller en `break` påträffats. Så följande exempel fungerar helt korrekt:

```
char Menyval;
cin » Menyval;
switch ( Menyval ) {
  case '1' : cout « "1 valdes" « endl;
             Index++;
             cout « "Index är nu " « Index « endl;
             break;
  case '2' : ...
  ...
}
```

Här kommer alla satser att utföras om `Menyval == '1'`. Låter det ologiskt kanske? Varför måste inte `switch` ha en blockkonstruktion när allting annat måste ha det? Svaret är att `switch` använder sig av en ganska praktisk funktionalitet som kallas *fall through*. Det betyder att så fort som en `case`-test är sann kommer dess sats(er) att utföras. Exekveringen kommer dock efter att `case`-kroppen utförts att direkt fortsätta med kroppen för nästa `case`-sats, d.v.s. exekveringen "faller igenom". För att stoppa detta fenomen kan man använda sig av `break`. Denna stoppar exekveringen av `switch`-konstruktionen direkt och fortsätter på satsen *efter* själv `switch`-blocket. Ett exempel då fall-through är praktiskt är då man vill göra en liten meny som accepterar både versaler och gemener (stora och små bokstäver). Istället för att duplicera koden för båda fallen låter man den ena falla igenom:

```
char Menyval;
cin » Menyval;
switch ( Menyval ) {
  case 'S' :
  case 's' : // spara en fil
             break;
  case 'l' :
  case 'L' : // ladda in en fil
```

```
            break;
  case 'q' :
  case 'Q' : // avsluta
            break;
  ...
  default : // felaktigt val!
            cout « "Felaktigt val!" « endl;
}
```

Här utnyttjas *fall through* på ett effektivt (och vackert!) sätt. I de flesta fall vill man dock inte ha denna effekt, så glöm inte bort att sätta en `break` där du inte vill ha *fall through*! Sist i konstruktionen ovan finns en `default`. Denna samlar upp alla värden på `Menyval` som inte matchades av någon `case`-sats. I vårt exempel har användaren då gett in ett menyval som inte kändes igen. Eftersom `break` används på alla ställen i de övriga `case`-satserna kan vi vara säkra på att alla giltiga val fångats upp och behandlats.

6.5. `? :`-operatorn

C++ innehåller ytterligare en konstruktion som kan användas ungefär på samma sätt som en `if`-sats. Denna är en relik från att C++ innehåller allt som C innehåller. Den allmänna formen för uttryck men `? :`-operatorn är:

```
uttryck1 ? uttryck2 : uttryck3;
```

Detta skall tolkas som att *uttryck1* först evalueras. Om denna evaluering ger värdet `true` får hela uttrycket värdet för *uttryck2*, i annat fall får hela uttrycket värdet för *uttryck3*. Ett exempel på hur man kan välja det mindre av två tal visas nedan:

```
// get Tal1 och Tal2 värden
int Tal1 = ...;
int Tal2 = ...;
int MindreTalet = Tal1 < Tal2 ? Tal1 : Tal2;
```

Här får `MindreTalet` värdet på `Tal1` om `Tal1 < Tal2`, i annat fall får `MindreTalet` värdet på `Tal2`. Detta kan verka lite kryptiskt, och det är det också.

Man bör vara försiktigt med att använda `? :`-operatorn, eftersom det är enkelt att göra fel. I vissa fall är den dock helt praktisk, speciellt när en normal `if`-sats skulle bli för lång och "skymma" den relevanta koden.

```
// get Tal1 och Tal2 värden
int Tal1 = ...;
int Tal2 = ...;
cout « "Tal1 är " (Tal1 < Tal2 ? "mindre" : "större")
     « " än Tal2." « endl;
```

Här används `? :`-operatorn för att välja mellan texten `"mindre"` och `"större"`, beroende på värdet på uttrycket `Tal1 < Tal2`. Parenteser har använts för att gruppera koden snyggare och göra den mera läslig.

Kapitel 7. Funktioner

Detta kapitel behandlar funktioner i C++. Funktioner är den grundläggande formen av modularisering i C++.

7.1. Grundbegrepp

I de flesta programmeringsspråk finns det olika sätt att ordna kod i olika separata block eller moduler som associeras med ett namn. Dessa brukar kallas *funktioner* eller *procedurer*, beroende på språket. Även i C++ finns det funktioner. De är den grundläggande byggstenen för modulariserade program och ger programmeraren de första möjligheterna att göra större programhelheter. En funktion i C++ kategoriseras av att den har ett *namn*, ett *returnvärde* (se Avsnitt 7.1.2) och ett antal *parametrar* (se Avsnitt 7.1.1). Den allmänna formen för en funktion ser ungefär ut så här:

```
returntyp funktionsnamn (parametertyp1 parameter1, ... ) {
    satser
}
```

Funktioner kan namnges precis på samma sätt som variabler. De kan innehålla alfabetiska tecken, siffror och _, och följer samma regler som variabler.

När det gäller funktioner är { och } obligatoriska för att avgränsa det som hör till funktionen. Likaså är en tom parentes `()` obligatorisk även om funktionen inta tar några parametrar.

7.1.1. Parametrar till funktioner

För att funktioner skall kunna göra någon nytta behöver de för det mesta ett antal parametrar som är det data de jobbar med. Följande enkla funktion tar inga parametrar överhuvudtaget, och skriver endast ut en enkel text:

```
void Print1 () {
```

```
    cout « "Hej från Print1!" « endl;
}
```

Ovanstående funktion tar inga parametrar, och dess parameterlista är således en tom parentes `()`. Vi kan ändra funktionen så att den tar en parameter av typen `int` som även skrivs ut:

```
void Print2 (int Tal) {
    cout « "Hej från Print2, talet är:" « Tal « endl;
}
```

Nu börjar funktionen redan göra någon nytta. Parametern `Tal` kan användas som en helt normal variabel av koden i `Print2`, d.v.s. den kan bl.a. tilldelas och skrivas ut. Låt oss modifiera funktionen så att den tar en valfri text som parameter, samt ett tal som anger hur många gånger texten skall skrivas ut:

```
void Print3 (string Text, int Tal) {
  int Index;
  for ( Index = 0; Index < Tal; Index++ ) {
    cout « Text « endl;
  }
}
```

Parametrar separeras alltså med ett kommatecken (`,`). För funktionens del är det ingen skillnad i vilken ordning parametrarna finns. Exemplet ovan introducerar även en *lokal variabel*, men vi kommer till det senare. `Index` används till att iterera `Tal` antal gånger och skriva ut `Text`. Låt oss nu göra en funktion som beräknar en rektangels area:

```
void RektangelArea1 (float Kant1, float Kant2) {
   // beräkna radien och skriv ut
   float Area = Kant1 * Kant2;
   cout « "Arean är: " « Area « endl;
}
```

Detta börjar redan närma sig en funktion som vi kan se en viss nytta med. En allvarlig brist är dock att vi inte hittills kan förmedla resultatet tillbaka till anroparen. Det är inget problem, läs vidare.

7.1.2. Returvärden

Funktioner är ganska värdelösa om de inte kan förmedla returvärden tillbaka till den som anropade funktionen. Redan namnet *funktion* implicerar ju att det är frågan om någonting som utför en operation och returnerar någonting. I exemplen tidigare användes inget returvärde, utan returtypen var deklarerad som `void`. Denna datatyp betyder att ingenting returneras, att returvärdet inte helt enkelt finns. Vi skall nu göra om vår areaberäknande funktion så att den returnerar ett värde:

```
float RektangelArea2 (float Kant1, float Kant2) {
   // beräkna radien och skriv ut
   float Area = Kant1 * Kant2;

   // returnera arean
   return Area;
}
```

Nu är returntypen definierad till `float`. Nytt är även att sist i funktionen utförs satsen `return`. Denna används för att förmedla ett returvärde, som placeras efter `return`. När denna sats exekveras avbryts funktionen omedelbart och exekveringen fortsätter där funktionen anropades. I funktioner som inte returnerar någonting alls kan `return` även användas för att avbryta exekveringen av funktionen, men då ges inget värde till `return`. Vi kan modifiera vår `Print3` och göra den lite mera robust:

```
void Print4 (string Text, int Tal) {
  int Index;
  // verifiera att Tal är > 0
  if ( Tal <= 0 ) {
    cout « "Aj, aj, kan inte skriva ut något..." « endl;
    return;
  }
  else {
    for ( Index = 0; Index < Tal; Index++ ) {
      cout « Text « endl;
    }
  }
}
```

Nu använder vi oss av en tom `return` ifall användaren gett in ett `Tal` som är `<=` 0. I så fall kan ju inget skrivas ut. Funktionen `Print4` har returndatatypen `void`, så det är ogiltigt att försöka returnera någonting. På samma vis bör en funktion som enligt sin definition skall returnera ett returvärde av en viss typ även göra det.

Man kan returnera olika typer av data, t.ex. variabler, konstanter o.s.v. Kravet är att den "mottagande" vaiabeln är av samma typ som den typ som funktionen returnerar, eller av en typ till vilken det går att konvertera automatiskt.

7.2. Anropa funktioner

Det är mycket enkelt att anropa funktioner i C++. Det allmänna beteckningssättet ser ut som följande:

```
funktion();
funktion(parametrar);
variabel = funktion();
variabel = funktion(parametrar);
```

Man kan alltså direkt anropa en funktion genom att skriva dess namn följt av en parentes med parametrarna. Om inga parametrar behövs ges en tom parentes. Om funktionen returnerar ett värde kan värdet sparas i en variabel eller användas "direkt". Exempelvis:

```
Print1 ();
Print3 ( "Hello world", 5 );
float Area = RektangelArea2 ( 4.0, 2.5 );
cout « "Arean är: " « RektangelArea2 ( 4.0, 2.5 ) « endl;
```

Funktioner är mycket enkla att använda i C++, och är även väldigt flexibla, som vi efterhand skall se.

7.3. Parametertyper

I de exempel vi hittills behandlat har den normala metoden för parameteröverföring använts, nämligen *värdeparametrar*. Detta betyder att då en funktion anropats med en parameter har endast variablens värde kopierats till funktionen. Den parameter som funktionen sedan använder har ingenting gemensamt med den variabel/konstant som värdet ursprungligen kom ifrån, den har helt enkelt kopierats.

```
#include <iostream>
void Test (int TestTal) {
  TestTal = 10;
  cout « "TestTal är: " « TestTal « endl;
}

int main () {
  int Tal = 5;

  cout « "Tal är: " « Tal « endl;
  Test ( Tal );
  cout « "Tal är: " « Tal « endl;
}
```

Man kunde lätt ledas att tro att funktionen `Test` ovan kan ändra värdet på parametern `TestTal` till 10, och samma värde skulle bli aktivt i `HuvudProgram`. Så är dock inte fallet, utan i `HuvudProgram` kommer `Tal` att fortfarande ha värdet 5. Vill vi att `Tal` skall ändra värde måste vi med våra kunskaper hittills returnera ett värde från `Test` och spara det i `Tal`. Körning av programmet ovan skulle ge:

```
% ParameterTest1
Tal är: 5
TestTal är: 10
Tal är: 5
%
```

7.3.1. Referensparametrar

Det finns ett alternativt sätt (egetligen två) att göra att `Test` kan ändra värdet på sin parameter så att det även reflekteras i den anropande funktionen. Denna funktionalitet åstadkomms genom s.k. *referensparametrar*, då man istället för en kopia av en variabel skickar en *referens* till en variabel till funktionen. För att göra en parameter till en referensparameter i en funktion räcker det med att skriva ett `&` mellan vaiabelns datatyp och namn i funktionens parameter lista. Vi kan nu skriva om ovanstående program så att det verkligen ändrar värdet på parametern.

```
#include <iostream>
void Test (int & TestTal) {
  TestTal = 10;
  cout « "TestTal är: " « TestTal « endl;
}

int main () {
  int Tal = 5;

  cout « "Tal är: " « Tal « endl;
  Test ( Tal );
  cout « "Tal är: " « Tal « endl;
}
```

Enda skillnaden i det modifierade exemplet är att vi har lagt till ett `&`-tecken. Detta kan göras för vilka datatyper som helst. Vi kommer att använda referensparametrar mera senare då vi behandlar klasser. Referensparametrar kan vara väldigt praktiska, t.e.x om man har en funktion som behöver returnera flera än ett värde. Då kan man skicka med variabler till funktionen dit resultatet placeras. Körning av programmet ovan skulle ge:

```
% ParameterTest2
Tal är: 5
TestTal är: 10
Tal är: 10
%
```

7.3.2. Konstanta parametrar

Ibland finns det behov för att kunna garantera att en parameter som skickas till en funktion inte ändras i funktionen. Speciellt då man använder sig av referensparametrar är det viktigt att kunna försäkra sig om att en viss funktion inte ändrar på värdet. Varför skickar man då data som referensparametrar, vorde det inte bättre att skicka den "helt normalt" i så fall? Orsaken (som vi skall se senare i Avsnitt 9.5.4) är effektivitet. Då man använder komplicerade datatyper är det mycket effektivare att skicka data som referensparametrar jämfört med att skicka dem som normala parametrar, eftersom mycket mindre data kopieras för funktionsanropet. I sådana fall vill man ha effektivitet, men även säkerhet. Detta kan uppnås genom att placera nyckelordet `const` framför parametern. I exemplet i förra kapitlet behövde vi ju egentligen inte manipulera parametern `TestTal` på något sätt, så då kunde vi ha gjort den till en konstant parameter istället:

```
void Test (const int & TestTal) {
  TestTal = 10;
  cout « "TestTal är: " « TestTal « endl;
}

void HuvudProgram () {
  int Tal = 5;

  cout « "Tal är: " « Tal « endl;
  Test ( Tal );
  cout « "Tal är: " « Tal « endl;
}
```

Enda skillnaden är ett extra `const`. Nu kommer kompilatorn att ge ett felmeddelande om man försöker ändra värdet på `TestTal` på något sätt. Senare då vi behandlar klasser (se Kapitel 14) kommer vi att använda `const` i mycket hög grad för att skydda parametrar.

Det kan anses vara god programmeringssed att använda `const` för parametrar som kan ändras av en funktion, men som funktionen inte har för avsikt att ändra. Det gör det lättare att läsa ett program då man snabbt kan se vilka parametrar som ändras av en funktion.

Om man har en konstant variabel (se Avsnitt 2.4) och denna skickas som referensparamameter *måste* man ha nycklordet `const` vid parameterdefinitionen, annars ger kompilatorn ett felmeddelande. Följande program är illegalt:

```
void testaKonstant (int & Tal) {
  // gör någonting
}

int main () {
  const int Test = 10;

  // anropa en konstant funktion
  testaKonstant ( Test );
}
```

Funktionen `testaKonstant()` har en referensparameter, och kan således ändra på värdet på `Tal`, men det tal som skickas är en konstant. I detta fall krävs således definitionen `const` för funktionsparametern, och `testaKonstant()` bör skrivas:

```
void testaKonstant (const int & Tal) {
  // gör någonting
}
```

Samma förfarande gäller all data som är konstant på ett eller annat sätt.

7.4. Funktionen `main()`

Som vi redan tidigare såg i programmet *Hello world!* finns det en speciell funktion som heter `main()`. Denna funktion är speciell på det sättet att den är alltid den första funktionen som utförs i ett C/C++-program. Denna funktion kan sedan i sin tur anropa andra funktioner och på så vis bygga upp den egentliga funktionaliteten i programmet. Det finns inga undantag till detta, `main()` är alltid den första funktionen. När funktionen `main()` avslutas, avslutas även hela programmet. Vi ska i fortsättningen alltid använda `main()` i exempel där det är möjligt, för att göra dem mera kompletta.

Funktionen `main()` kan inte heta något annat än just `main()`, inga andra stavningar tillåts. Om kompilatorn inte hittar denna funktion när den kompilerar ett program fås en varning eller ett fel. Ett program med två funktioner som anropas från `main()` kan se ut som följande:

```
#include <iostream>

float RaknaArea (float Sida1, float Sida2) {
  // beräkna och returnera en area av en rektangel
  return Sida1 * Sida2;
}

void SkrivSvar (float Area) {
  cout « "Arean är: " « Area « endl;
}

int main () {
  // beräkna arean
  float Area = RaknaArea ( 5.5, 10 );

  // skriv ut svaret
  SkrivSvar ( Area );
}
```

Här har vi nu två skilda funktioner som anropas från `main()`. En liten förenklig som kunde göras i `main()` är att kombinera areaberäkningen med utskriften. Eftersom `RaknaArea()` returnerar en `float`, och `SkrivSvar()` vill ha en `float` som parameter kan man skriva om `main()` på detta sätt:

```
int main () {
  // beräkna arean och skriv ut svaret
  SkrivSvar ( RaknaArea ( 5.5, 10 ) );
}
```

Nu använder vi resultatet från `RaknaArea()` direkt och skickar det till `SkrivSvar()` utan använda en temporär variabel som mellanlagring.

7.4.1. Returvärde från `main()`

Som vi sett deklareras `main()` alltid med returvärdet `int`, även om vi aldrig utför någon `return`-sats. Detta returvärde skall om det ges vara ett positivt heltal som skall ange hur programmets körning lyckades. Värdet 0 indikerar en lyckad körning utan fel, medan andra värden indikerar fel av olika typ. Det system som kör programmet kan således få reda på hur programmets exekvering gick. Om inget returvärde ges explicit placera kompilatorn automatiskt en sats `return 0` sist i programmet. Alltså om man inte manuellt specifierare returvärdet från `main()` fås alltid värdet 0. Detta gäller dock endast funktionen `main()`.

7.4.2. Prototyper

I exemplen ovan har vi alltid haft `main()` sist i programmet. Vad händer om man istället flyttar `main()` först, och placerar `RaknaArea()` och `SkrivSvar()` efter? Man skulle ju tycka att det inte vore någon skillnad överhuvudtaget, men det är tyvärr en skillnad. C++-kompilatorer är för det mesta ganska lata, och kräver att alla symboler, variabler och funktioner måste vara definierade före de används. Då kompilatorn läser ett program med `main()` först, kommer den att stöta på funktionerna `RaknaArea()` och `SkrivSvar()` som inte ännu är definierade, eftersom de definieras först några rader senare! Allting måste alltså vara definierat före det används. Tänk om man har en situation med två funktioner som anropar varandra. I det fallet går det helt enkelt inte att definera båda funktionerna före de används. Det finns en omväg runt detta dilemma, och det är något som kallas *prototyper*. En prototyp är en en definition som är menad för kompilatorn och berättar hur en funktion ser ut. Prototypen innehåller ingen kod, endast funktionens namn, returvärde och parametrar. Allmänt ser det ut så här:

```
returtyp funktionsnamn (parametertyp1 parameter1, ... );
```

En funktionsprototyp har ingen kropp innesluten mellan { och }. När kompilatorn stöter på en dylik rad vet den hur en viss funktion ser ut, och kan använda den redan innan den definieras. Kompilatorn vet även att själva koden för funktionen "kommer senare". Vi kan nu skriva om vår areaberäkning med prototyper så att `main()` är först:

```
#include  <iostream>

// prototyper för våra två funktioner
float RaknaArea (float Sida1, float Sida2);
void SkrivSvar (float Area);

int main () {
  // beräkna arean
  float Area = RaknaArea ( 5.5, 10 );

  // skriv ut svaret
  SkrivSvar ( Area );
}

float RaknaArea (float Sida1, float Sida2) {
  // beräkna och returnera en area av en rektangel
  return Sida1 * Sida2;
}

void SkrivSvar (float Area) {
  cout « "Arean är: " « Area « endl;
}
```

Notera de två prototyperna överst. De *måste* stämma överens med de parametrar och returvärden som sedan egentligen används av funktionerna. Om prototyper ges för funktioner som sedan aldrig skrivs kommer kompilatorn att ge ett felmeddelande. Den "magiska" raden `#include <iostream>` kommer att behandlas senare (se Kapitel 10), och nu räcker det med att raden syftar till en fil som innehåller prototyp-deklarationer för de I/O-funktioner vi använder (`cout`).

7.5. Parametrar till `main()`

Eftersom `main()` är en funktion borde den väl också kunna ta parametrar? Vilka är de i så fall, och var kommer de ifrån? Man kan faktiskt ge parametrar till `main()`, men de ges inte från någon anropande funktion, utan de kommer från de

kommandoradparametrar som användaren ger. Denna funktionalitet gör det möjligt att ge parametrar till våra program. I de exempel vi hittills visat där `main()` förekommit har inga argument till programmen använts, och således heller inga parametrar till `main()`. Vill man kunna behandla även dessa bör man sätta till två parametrar til sin `main()`-deklaration, nämligen en `int` som kommer att innehålla antalet parametrar, och en *strängvektor* som innehåller de givna argumenten. Vi kommer att behandla vektorer i Kapitel 9. Nedan kommer dock ett exempel som visar hur ett program kan se hur många parametrar det fått, samt skriva ut dessa:

```
#include  <iostream>
#include  <string>

int main (int argc, char * argv[]) {
  string Parameter;

  // hur många parametrar har vi fått?
  cout « "Vi har " « argc « " parametrar." « endl;

  // iterera över alla parametrar
  for ( int Index = 0; Index < argc; Index++ ) {
    // konvertera till en sträng och skriv ut
    Parameter = argv [ Index ];
    cout « "Parameter " « Index « " = " « Parameter « endl;
  }
}
```

Ovan ser du de två parametrarna `int argc` och `char * argv[]`. Dessa namn är de klassiska C-namnen på dessa parametrar, och det är bra att använda dessa så känns de lätt igen. Namnet `argc` kommer från *argument count*, och `argv` från *argument vector*. Den senare har datatypen `char *`, vilket är C:s sätt att representera strängar. Detta behandlas i Appendix A. Hakparentesen efter `argv` indikerar att det är frågan om en *vektor*. Denna innehåller `argc` antal värden av typ `char *`. Vi konverterar dessa till C++-strängar för att de är lättare att hantera, samt skriver ut dessa. En körning av programmet ovan kunde t.ex. ge följande resultat:

```
% ./CommandlineArguments1 1 -2 3.001 foo bar "Hello world"
Vi har 7 parametrar.
Parameter 0 = ./CommandlineArguments1
Parameter 1 = 1
Parameter 2 = -2
Parameter 3 = 3.001
Parameter 4 = foo
Parameter 5 = bar
Parameter 6 = Hello world
```

```
%
```

Här ser vi att programmet fick sex parametrar förutom sitt eget filnamn. Intressant är närmast den sista parametern. Här har vi två ord i en enda parameter. Detta fungerar tack vare att orden är skrivna inom `""`, varvid de tolkas som en enda parameter.

Om du kör programmet kommer du att märka att den första parametern alltid är programmets filnamn. Så även om du startar programmet helt utan parametrar så kommer alltid en att skickas till `main()`, nämligen namnet. Denna skickas alltid med så att programmet kan "veta vad det heter". Vi kunde modifiera ovanstående program en aning:

```
#include  <iostream>
#include  <string>

int main (int argc, char * argv[]) {
  string Parameter;
  string Namn;

  // namnet på programmet
  Namn = argv [0];

  // hur många parametrar har vi fått?
  cout « "Programmet '" « Namn « "' har " « argc - 1 « " parametrar." « endl;

  // iterera över alla parametrar förutom filnamnet
  for ( int Index = 1; Index < argc; Index++ ) {
    // konvertera till en sträng och skriv ut
    Parameter = argv [ Index ];
    cout « "Parameter " « Index « " = " « Parameter « endl;
  }
}
```

Vi vet alltid att `argv[0]` är programmets namn, och alla övriga värden är parametrar givna av användaren. Eftersom alltid minst en parameter skickas kan vi skriva kod som den ovan. Körning av programmet kunde t.ex. ge följande resultat:

```
% ./CommandlineArguments2 10 20 Hej
Programmet './CommandlineArguments2' har 3 parametrar.
Parameter 1 = 10
Parameter 2 = 20
Parameter 3 = Hej
%
```

Vi kan även enkelt göra att program som kräver ett visst antal parametrar, t.ex. namn på datafiler:

```
#include  <iostream>
#include  <string>

int main (int argc, char * argv[]) {
  string Parameter;
  string Namn;

  // har vi fått tillräckligt med parametrar?
  if ( argc != 2 ) {
// fel antal parametrar, skriv kort hjälptext och avsluta
cout « "Fel antal parametrar!" « endl;
cout « "Användning: " « argv[0] « " filnamn" « endl;
return 1;
  }

  // korrekt antal parametrar eftersom vi kommit ända hit
  cout « "Antalet parametrar korrekt." « endl;
}
```

Programmet ovan kommer att kräva ett visst antal parametrar från användaren (1). Men eftersom det alltid finns programmets namn med som den första parametern måste vi jämföra med 2. Här ser vi för första gången hur man kan returnera från `main()`. Vi har en `return 1` i `if`-satsens kropp. Denna kommer att avlsuta programmet med returvärdet 1. Vart går då detta returvärde? Jo, det förmedlas tillbaka till kommandotolken, som kan använda detta värde för att kontrollera om programmet utfördes korrekt eller inte (se Avsnitt 7.4.1). Normalt låter vi kompilatorn returnera 0 om allt gick rätt till, men eftersom programmet ovan fick fel antal parametrar returnerar vi 1 istället för 0. Körning av programet ovan kunde t.ex. ge:

```
% ./CommandlineArguments3
Fel antal parametrar!
Användning: ./CommandlineArguments3 filnamn
% ./CommandlineArguments3 foobar
Antalet parametrar korrekt.
%
```

7.6. Omfattning (scoping)

Med *scoping* avses det område av programkoden där en viss variabel är definierad. En variabel kan inte tilldelas eller refereras på ett ställe i koden där den inte är definierad. Scoping-reglerna för C++ är ganska enkla, men det finns några små fällor.

7.6.1. Lokala variabler

Normalt deklareras variabler i C++ i en viss funktion. Dessa variabler är då *lokala* för denna funktion, och deras scope är således endast denna funktion. Detta gäller även andra block av kod, t.ex. kan man ha lokala variabler i en `if`-sats. De flesta variabler är lokala variabler. Alla variabler som vi använt hittills i våra exempel är lokala variabler. Ett exempel:

```
void exempel (int Parameter) {
  int Hjalp;
}

int main () {
  int Tal;
  float Area;

  // anropa vår funktion
  exempel ( Tal );
}
```

Det enkla "programmet" ovan har tre lokala variabler, nämligen `Tal`, `Area`, `Hjalp` och `Parameter`. De har alla scope endast i sina respektive funktioner, d.v.s. `exempel()` kan inte referera varken `Tal` eller `Area`. Vill vi att en funktion skall kunna referera en viss lokal variabel måste den skickas med som parameter till funktionen i fråga. Även om två funktioner har lokala variabler med samma namn förekommer inga problem, de är helt enkelt skilda variabler med olika scope. Om man i misstag har både en lokal variabel och en parameter med samma namn bör kompilatorn ge ett felmeddelande. Lokala variabler är enkla och naturliga att använda, och vi kommer att använda dem även i alla exempel framöver.

7.6.2. Globala variabler

Motsatsen till lokala variabler är *globala variabler*. En global variabel har scope i hela den fil där den är deklarerad (och även i andra om den definieras så). Detta innebär i praktiken att alla funktioner kan referera den globala variabeln utan att behöva deklarera detta på något sätt. Ett exempel:

```
#include  <iostream>

// global variabel
int Global = 10;

void testaGlobal () {
  // sätt ett nytt värde på den globala variabeln
  Global = 20;
}

int main () {
  cout « "Global har värdet: " « Global « endl;
  testaGlobal ();
  cout « "Global har värdet: " « Global « endl;
}
```

Om vi kör ptogrammet ser vi att `Global` först har värdet 10 och efter att `testaGlobal()` exekverats har den värdet 20. Globala variabler måste deklareras före de kan användas. Om vi har ett program som är utspritt över flera filer och vi har en global variabel i en fil `A` som vi vill referera till i en annan fil `B`, måste vi deklarera variabeln som `extern` i `B` så att kompilatorn vet att det är en global variabel som är definierad i en annan fil. Kör man ovannämnda program får man denna utskrift:

```
% ./GlobalVariable
Global har värdet: 10
Global har värdet: 20
%
```

7.6.3. Statiska variabler

Normala lokala variabler håller sitt värde så länge som de är i scope, d.v.s. normalt så länge den funktion där de är definierade exekveras. Så fort som funktionen eller kodblocket avslutas tappar variabeln sitt värde för att åter bli nollställd nästa gång funktionen anropas. Globala variabler däremot håller sitt värde under hela programmets exekveringstid. Ibland kanske man dock vill ha en lokal variabel som kan hålla sitt värde mellan anrop av funktionen. Man kanske vill hålla ordning på en viss flagga eller något beräknat värde. I dessa fall kan man använda sig av *statiska* variabler. Dessa definieras genom att sätta ordet `static` framför variablens datatyp då den definieras. Allmänt ser det ut så här:

```
static datatyp variabelnamn;
```

Ett enkelt exempel på en funktion som håller reda på hur många gånger den anropats kan man göra på följande sätt:

```
#include  <iostream>

void raknaAnrop () {
  // statisk variabel med värdet 0
  static int Anrop = 0;

  // öka antalet anrop med 1
  Anrop++;
  cout « "raknaAnrop har anropats: " « Anrop « " gånger." « endl;
}

int main () {
  // anropa funktionen 'Anrop' några gånger
  raknaAnrop ();
  for ( int Index = 0; Index < 5; Index++ ) {
    raknaAnrop ();
  }
}
```

Vi initierade `Anrop` till 0 på samma gång som vi deklarerade variabeln. Vill man ha ett intitialvärde på en statisk variabel måste det göras på detta sätt, alltså samtidigt som deklarationen.

Ofta används statiska variabler av typen `bool` för att hålla reda på om t.ex. en fil är öppnad, någonting speciellt redan initierats o.s.v. Det kan i så fall se ut som nedan:

```
void lasData () {
  static bool Initierad = false;

  // har vi redan initierat någon resurs, t.ex. en fil?
  if ( ! Initierad ) {
    // initiera resursen
    ...
    // nu har vi initierat resursen
    Initierad = true;
  }

  // gör något med resursen t.ex. läs data
}

int main () {

  // läs data i alla evighet
  while ( true ) {
    lasData ();
  }
}
```

Här har vi nu en boolesk flagga som vi använder för att veta om vi redan initierat någon viss resurs som endast skall initieras en gång. När `lasData()` anropas första gången har `Initierad` värdet `false`, och `if`-satsen utförs således. Efter detta har vi utfört initieringen och flaggan kan sättas till `true`, och `if`-satsen körs aldrig mera.

7.7. Rekursion

Rekursion är ett begrepp som brukar förknippas med funktionell programmering. Trots att C++ är ett imperativt språk kan man även här använda rekursion för att lösa olika problem. Rekursion innebär som bekant att funktioner kan anropa sig själv. Rekursion är inte alltid det mest optimala förfaringsättet i C++, men i vissa fall är det väldigt praktiskt, och kan t.o.m. vara effektivare on iterativ kod. T.ex. vid hantering av *träd* av olika typ är rekursion en ovärderlig resurs. Ett standardexempel på rekursion torde vara hur `n!` (n-fakultet) räknas ut rekursivt:

```
#include <iostream>

unsigned int Fakultet (unsigned int N) {
  if ( N == 0 ) {
    // basfallet nått
    return 1;
  }

  return N * Fakultet ( N - 1 );
}

int main () {
  unsigned int Tal;
  cout « "Ge in ett tal: ";
  cin » Tal;

  // beräkna fakulteten och skriv ut svaret
  cout « Tal « "! = " « Fakultet ( Tal ) « endl;
}
```

Man bör dock se upp för oändlig eller alltför djup rekursion, eftersom det leder till problem med programmets stack, vilket i sin tur leder till problem i exekveringen. Ett annat exempel på rekursiv programmering är beräkning av *Fibonacci-tal*. Fibonacci-talet `fib(n)` definieras som `fib(n-1) + fib(n-2)`, där `fib(0) = 0` och `fib(1) = 1`. Vi kan nu använda denna definition för att skapa en enkel implementation.

```
#include <iostream>

unsigned int fib (unsigned int N) {
```

```
  if ( N == 0 ) {
    // basfall 1 nått
    return 0;
  }

  if ( N == 1 ) {
    // basfall 2 nått
    return 1;
  }

  // almänt fall, använd rekursion
  return fib ( N - 1 ) + fib ( N - 2 );
}

int main () {
  unsigned int Tal;
  cout « "Ge in ett tal: ";
  cin » Tal;

  // beräkna fakulteten och skriv ut svaret
  cout « Tal « "! = " « fib ( Tal ) « endl;
}
```

7.8. Funktioner som parametrar

En aspekt av C som inte används speciellt ofta i C++ är möjligheten att skicka funktioner som parametrar till andra funktioner. Detta möjliggör en form av generisk programmering där man kan skapa funktioner vars beteende kan ändras genom att olika funktioner skickats som parameter applicaeras på någon form av data.

I C++ kan man för det mesta använda andra metoder för att uppnå samma funktionalitet, t.ex. med hjälp av klasser (se Kapitel 14).

7.8.1. Funktionsvariabler

Man kan skapa variabler som innehåller *adressen* för en funktion. Dessa variabler kan sedan skickas som parametrar till andra funktioner, för att sedan i något skede anropas. Allmänt definierar man en variabel som kan "innehålla" eller peka på (se Kapitel 9) en funktion på följande sätt:

```
returtyp (*variabelnamn) (parametertyp1, parametertyp2, ...);
```

Variabeln måste vara definierad exakt på samma sätt som den funktion man vill att den skall kunna innehålla. Vill man ha en variabel som innehåller en funktion som returnerar en `int` och tar som parameter två `string` skall den definieras på följande sätt:

```
int (*variabelnamn) (string, string);
```

Ifall funktionen inte returnerar någonting skall dess returtyp vara `void`, och ifall den inte tar några parametrar skall parameterlistan vara tom. Parentesen och asterisken (`*`) runt variabelnamnet indikerar att det är frågan om en pekare till en funktion.

Innan en funktionsvariabel kan användas måste den tilldelas adressen för en funktion. Vi kan t.ex. ha en funktion som tar en `int` som argument och returnerar en annan `int`. Vi kan då tilldela denna funktion till en funktionsvariabel på följande sätt:

```
// funktion som dubblerar ett tal
int dubblera (int Tal) {
  return Tal + Tal;
}

// variabel för funktion
int (*funktion)(int);

// tilldela variabeln
funktion = dubblera;
```

En alternativ syntax för själva tilldelningen är:

```
// tilldela variabeln
funktion = &dubblera;
```

Där ger man explicit adressen för funktionen. Båda sätten är ekvivalenta.

7.8.2. Anropa funktionsvariabler

Om man har en variabel som innehåller en funktion kan den användas som vilken normal funktion som helst. Om vi ser på exemplet ovan med en variabel som heter `funktion` kan den anropas på följande sätt:

```
// anropa funktionsvariabel
int Resultat = funktion (10);
```

Man säger att man *avrefererar* funktionsvariabeln. Ett lite mera omfattande exempel på hur funktionsvariabler kan användas kommer nedan.

```
#include <iostream>
#include <iomanip>

// funktion som dubblerar ett tal
int dubblera (int Tal) {
  return Tal + Tal;
}

// funktion som kvadrerar att tal
int kvadrera (int Tal) {
  return Tal * Tal;
}

// funktion som applicerar en given funktion på en serie med tal
void applicera (int (*funktion)(int) ) {
  // iterera över talen 0 till 4
  for (int Index = 0; Index <= 4; Index++ ) {
    // applicera den givna funktionen på talet
    cout « "Tal: " « setw(3) « Index « " blir: "
         « setw(4) « funktion(Index) « endl;
  }
}

int main () {
  cout « "Funktion: dubblera" « endl;

  // anropa applicera() med funktionen dubblera()
  applicera ( dubblera );

  cout « endl « "Funktion: kvadrera" « endl;

  // anropa applicera() med funktionen kvadrera()
```

```
  applicera ( kvadrera );
}
```

Här skickas funktionerna `dubblera()` och `kvadrera()` som parametrar till `applicera()` i tur och ordning. Inne i `applicera()` används parametern `funktion` som vilken normal funktion som helst som tar som parameter en `int` och returnerar en `int`. På detta sätt kan man skapa generiska funktioner som utför någon viss beräknin på en serie tal.

Notera att då vi skickar funktionerna som parametrar skriver vi *inte* ut tomma paranteser efter namnen, utan endast funktionens namn.

Körning av programmet ovan ger följande utskrift:

```
% ./FunktionsParameterar
Funktion: dubblering
Tal:   0 blir:    0
Tal:   1 blir:    2
Tal:   2 blir:    4
Tal:   3 blir:    6
Tal:   4 blir:    8

Funktion: kvadrering
Tal:   0 blir:    0
Tal:   1 blir:    1
Tal:   2 blir:    4
Tal:   3 blir:    9
Tal:   4 blir:   16
%
```

Kapitel 8. Input och output

Detta kapitel redogör för hur grundläggande I/O fungerar i C++. Läsaren lär sig hur data kan skrivas ut till skärmen och läsas in från tangentbordet. Trots att de enklaste formerna redan behandlats i Kapitel 4 tas allting upp från grunderna här för att få all information sammanfattad på ett ställe.

8.1. Streams och buffring

Nästan all I/O i C++ går via något som brukar kallas *streams* på engelska, ungefär "strömmar" på svenska. Dessa syftar till att data som t.ex. skrivs ut till skärmen hamnar i den stream som går till skärmen. En stream är en serie med bytes. I program som skriver ut text på skärmen kan man se denna ström av bytes som en ström av tecken, där varje byte är ett tecken. Förutom strömmen som går ut ur programmet till skärmen finns det även andra typer av strömmar. Det finns bl.a. en ström som innehåller all indata som kommer från tangentbordet, från vilken programmet kan läsa kommandon från användaren. Även filhantering bildar strömmar i C++.

8.1.1. Buffring av data

All data som skrivs till eller läses från de normala strömmarna är *buffrad*. Detta har endast ringa betydelse för programmeraren, men behandlas här för att ge bakgrundsinformation. Då ett program läser eller skriver till t.ex. en fil eller skärmen kommer all data som finns i strömmarna att vara buffrad. Denna buffring sköts av operativsystemet och är till för att göra I/O mera effektiv. Genom att buffra data kan man samla ihop större mängder data som sedan snabbt skrivs på en gång till disk. Samma händer med text som skrivs ut på skärmen, den blir även buffrar, för att sedan skrivas ut på en gång. Då ett program läser data från en fil läser operativsystemet in en större mängd av filen på en och samma gång och buffrar sedan det lästa. Mera om buffring kan du läsa t.ex. i kursen om operativsystem. Det är dock viktigt att veta att buffring förekommer, eftersom den i vissa fall inverkar på vad som t.ex. syns på

skärmen.

8.2. Utskrift till skärmen

Som vi redan sett i olika exempel så sköts utskrifter till skärmen i C++ på ett sätt som verkar lite lustigt. Man har använt en variabel eller liknande som heter `cout` och skickat data till denna m.h.a. «. `cout` är faktiskt en global variabel för varje program som har raden

```
#include  <iostream>
```

någonstans bland övriga inkluderade filer (se Kapitel 10 för mera information). Denna variabel är programmets stream ut till skärmen. Man brukar prata om att `cout` är programmets *standard output*. Egentligen är `cout` en instans av klassen `ostream`, vilket kommer att förklarar dess konstiga beteende, men det behandlas senare.

Den enda uppgiften `cout` har är att skriva ut variabler, konstantet och annan data till den normala ut-strömmen. Som vi kommer att se är `cout` väldigt bra på det den gör, och möjliggör mycket flexibla konstruktioner. Den allmänna formen för att skriva ut någonting till `cout` (och följdaktigen skärmen i normala fall) är:

```
cout « konstant;
cout « variabel;
cout « variabel1 « variabel2 « ... « variabelN
```

Tecknen « används för att "styra" data till`cout`, så det som är till höger om « kommer att skrivas ut. Som vi redan tidigare sätt kan man konkatenera flera utskrifter samtidigt, d.v.s. placera flera « efter varandra med variabler eller konstanter emellan. Några exempel på utskrifter:

```
int Tal;
float Tid;
string Svar;
cout « "Hello world!";
cout « Tal;
```

```
cout « "Svaret är: " « Svar;
cout « "Ping-tiden = " « Tid « " millisekunder";
```

Man kan alltså kombinera ihop olika datatyper vid utskrifter. `cout` är så flexibel att den kan hantera alla primitiva datatyper (se Kapitel 2) samt även strängar. Dessa kan blandas helt fritt. Man kan även formatera placeringen av de utskrivna variablerna precis som man själv vill. Följande är helt korrekt:

```
float Tid;
cout « "Ping-tiden = "
     « Tid
     « " millisekunder";
```

Det finns alltså ingenting som hindrar att utskriftskoden görs läslig och lättförståelig.

8.2.1. Buffring och `cout`

Även data skickat till `cout` kommer att buffras. Ibland kan data som skrivs till `cout` verka "fastna" i de interna buffrarna och kommer inte ut på skärmen med en gång. I vissa fall kan all utskrift från ett kort program komma först då programmet terminerar, vilket troligtvis inte är vad som avses. Man kan tvinga `cout` att tömma sina buffrar på två sätt. Det ena och mera använda sättet har vi redan använt i våra exempel. Det fungerar så att man skriver ut en symbol `endl` (end line) till sist. Denna åstadkommer att radbyte och även en tömning av buffern. Efter en `endl` kommer nästa utskrift att börja längst till vänster på nästa rad. Exempelvis:

```
float Tid;
string Svar;
cout « "Svaret är: " « Svar « endl;
cout « "Ping-tiden = " « Tid « " millisekunder" « endl;
```

Om ingen `endl` skrivs ut kommer nästa utskrift att följa direkt där den senaste slutade. I vissa fall kanske man vill tömma buffern men *inte* byta rad. I så fall kan man skriva ut en `flush`. Denna tömmer buffern. Att skriva ut en `endl` betyder att en osynlig `flush` även skrivs ut följt av ett radbyte.

8.2.2. Utskrift i olika talbaser

Det finns en hel del funktioner som kan användas för att formatera den utskrift som `cout` gör. I normala fall då man skriver ut heltal vill man få den normala representationen i basen 10, men ibland kanske man vill skriva ut ett tal i basen 8 eller 16. Exemplet nedan visar hur detta kan göras:

```
#include <iostream>

int main () {
  int Tal = 167;

  // skriv ut normalt
  cout « Tal « endl;

  // hexadecimalt
  cout « hex « Tal « endl;

  // oktalt
  cout « oct « Tal « endl;

  // decimalt igen
  cout « dec « Tal « endl;
}
```

Körning av programmet ger utskriften:

```
% ./Output2
Dec: 167
Hex: a7
Oct: 247
Dec: 167
%
```

Här skrivs `Tal` ut först på normalt sätt, sedan hexadecimalt, därefter oktalt och till sist decimalt igen. Omvandlingen mellan dessa sker via något som kallas *manipulatorer*. I exemplet används manipulatorerna `dec`, `hex` och `oct`. Dessa kan precis som `flush` och `endl` skrivas ut till en ström. De får alla efterföljade heltal av olika typer att skrivas ut med den givna basen, och är i kraft till en ny bas sätts. Ett alternativt sätt att åstadkomma samma effekt är:

```
#include <iostream>

int main () {
  int Tal = 167;

  // skriv ut normalt
  cout « Tal « endl;

  // hexadecimalt
  hex ( cout );
  cout « Tal « endl;

  // oktalt
  oct ( cout );
  cout « Tal « endl;

  // skriv ut normalt igen
  dec ( cout );
  cout « Tal « endl;
}
```

Här används funktionerna `dec()`, `hex()` och `oct()` för att ändra basen på strömmen `cout`, som ges som parameter. Båda sätten är ekvivalenta, och vilken man använder är en smakfråga. Det finns även en tredje metod som använder sig av en mekanism som vi inte ännu behandlat, nämligen *metoder*. Du kan tänka dig metoder som funktioner associerade med en viss variabel, i detta fall `cout`. Vi använder oss av metoden `cout()` för att ställa vilken talbas som används. Samma exempel ovan kan då skrivas:

```
#include <iostream>

int main () {
  int Tal = 167;

  // skriv ut normalt
  cout « Tal « endl;

  // hexadecimalt
  cout.setf ( ios::hex, ios::basefield );
  cout « Tal « endl;
```

```
  // oktalt
  cout.setf ( ios::oct, ios::basefield );
  cout « Tal « endl;

  // skriv ut normalt igen
  cout.setf ( ios::dec, ios::basefield );
  cout « Tal « endl;
}
```

De två argument som ges till `setf()` är först en flagga som anger vilket värde som skall ändras och det andra anger "området", i detta fall `ios::basefield`, som kontrollerar använd talbas.

8.2.3. Utskrift av flyttal

Att skriva ut flyttal på skärmen kan vara en aning problematiskt. Man kan vilja justera antalet decimaler som visas, om nollor skall visas eller om man skall använda "vetenskaplig" notation på utskriften o.s.v. C++ erbjuder oss rika möjligheter att justera alla dessa parametrar då vi skriver ut flyttal m.h.a. `cout`. För att precisera om tal skall skrivas ut i "vetenskaplig" notation, decimalnotation eller blandad notation använder vi igen metoden `setf()` hos `cout`. Det hela ser ut som följande:

```
#include <iostream>

int main () {
  double Tal = 1234567.89;

  // normal notation
  cout « Tal « endl;

  // alltid med decimalkomma
  cout.setf ( ios::fixed, ios::floatfield );
  cout « Tal « endl;

  // vetenskaplig notation
  cout.setf ( ios::scientific, ios::floatfield );
```

```
  cout « Tal « endl;
```

Kör vi detta program får vi resultatet:

```
% Output3
1.23457e+06
1234567.000000
1.234567e+06
1.23457e+06
%
```

Man anropar alltså funktionen genom att sätta en punkt (.) mellan `cout` och `setf()`. Mera om denna notation i Kapitel 14. Som parameter tar `setf()` en konstant (jo, det är en konstant, mer om detta i Kapitel 9) som igen säger vilken typ av notation som önskas och en som säger vilket "område" vi ändrar. Alternativen är

- `ios::fixed` skriver alltid ut talen med en decimalpunkt, oberoende av storlek eller antal decimaler.
- `ios::scientific` skriver ut talen i vetenskaplig notation med ett decimaltal följt av ett *e* och positiv eller negativ exponent.
- `0` som är det normala sättet. Tal skrivs då ut som en blandning av de ovanstående beroende på talet som skall skrivas ut. Denna notation är standard.

Förutom att notationen kan ändras kan man med konstanten `ios::showpoint` tvinga `cout` att visa ett decimalkomma efter talen även då det inte finns några decimaler. Man kan även välja antalet siffror som skall visas. Detta kan göras med *manipulatorn* `setprecision`, som tar en parameter som anger antalet siffror. I notationerna `ios::fixed` och `ios::scientific` betyder detta tal antalet decimaler, inte totala antalet tal. Ett exempel på ett program som beräknar kvadratroten och fjärderoten på alla tal mellan 1 och 10:

```
#include <iostream>
#include <iomanip>
#include <math.h>

int main () {
```

```
  double Rot;

  // alla tal mellan 1 och 10
  for ( int Index = 1; Index <= 10; Index++ ) {
    // beräkna roten
    Rot = sqrt ( Index );

    // skriv ut med två decimaler
    cout « Index « ": " « setprecision ( 2 ) « Rot;

    // beräkna fjärderoten
    Rot = sqrt ( Rot );

    // skriv ut med fyra decimaler
    cout « ", " « setprecision ( 4 ) « Rot « endl;
  }
}
```

Kvadratroten visas med endast två siffror, medan fjärderoten visas med 4:

```
% SqareRoot
1: 1, 1
2: 1.4, 1.189
3: 1.7, 1.316
4: 2, 1.414
5: 2.2, 1.495
6: 2.4, 1.565
7: 2.6, 1.627
8: 2.8, 1.682
9: 3, 1.732
10: 3.2, 1.778
%
```

8.2.4. Formatering av utskrifter

Förutom de ovannämnda formateringsmöjligheterna som gäller för flyttal finns det även en del allmänna formateringsmöjligheter i C++. En del av dessa är

- `ios::showbase` gör att `cout` explicit skriver ut basen för varje utskrivet tal. Hexadecimala tal får prefixet `0x` och oktala tal prefixet `0`.
- `ios::showpos` gör att `cout` placerar ett + framför positiva tal.
- `ios::uppercase` som tvingar hexadecimal och vetenskaplig utskrift att använda versaler.

Dessa ges även med `setf()`, men med den skillnaden att inget "område" ges. Exempelvis:

```
int Tal = 64;

// tvinga visning av notation
cout.setf ( ios::showbase );
cout « "Tal = " « hex « Tal « " (hex)" « endl;
```

Ofta vill man kunna påverka den bredd en utskrift får. Då det är frågan om t.ex. strängar är bredden vanligen strängens längd, men då man skriver ut tal av olika typ vill man ibland kunna ordna dem i kolumner eller på något annat sätt kontrollera bredden. Detta kan åstadkommas med metoden `width()` för `cout`. Som parameter ges ett tal som anger bredden i tecken. Denna bredd gäller för endast nästa utskrivna variabel eller konstant. Bredden 0 är standard och betyder att `cout` använder så många tecken som behövs. Vi kan avsåledes skriva om vårt rot-exemepel från ovan enligt:

```
#include <iostream>
#include <iomanip>
#include <math.h>

int main () {
  double Rot;

  // alla tal mellan 1 och 10
  for ( int Index = 1; Index <= 10; Index++ ) {
    // beräkna roten
    Rot = sqrt ( Index );

    // skriv ut med två decimaler
    cout.width ( 2 );
    cout « Index « ": ";
```

```
    cout.width ( 4 );
    cout « setprecision ( 2 ) « Rot;

    // beräkna fjärderoten
    Rot = sqrt ( Rot );

    // skriv ut med fyra decimaler
    cout « ", ";
    cout.width ( 6 );
    cout « setprecision ( 4 ) « Rot « endl;
  }
}
```

Notera de två `cout.width (n)` i exemlet ovan. Notera även att trots att en breddbestämmelse gäller endast nästa utskrivna tal/konstant så är en `setprecision(n)` ingen utskrift, utan bredden påverkar först variabeln `Rot`. Det kan ibland kännas klumpigt att använda `width()` för att justera bredden. Då kan man använda manipulatorn `setw()` istället, som fungerar på samma sätt som `setprecision()`. Kör vi programmet får vi följande utskrift:

```
% SqareRoot2
 1:    1,      1
 2:  1.4,  1.189
 3:  1.7,  1.316
 4:    2,  1.414
 5:  2.2,  1.495
 6:  2.4,  1.565
 7:  2.6,  1.627
 8:  2.8,  1.682
 9:    3,  1.732
10:  3.2,  1.778
%
```

Exemplet ovan blir då en aning snyggare och ser ut så här:

```
#include <iostream>
#include <iomanip>
#include <math.h>
```

```
int main () {
  double Rot;

  // sätt decimalnotation
  cout.setf ( ios::fixed, ios::floatfield );

  // alla tal mellan 1 och 10
  for ( int Index = 1; Index <= 10; Index++ ) {
    // beräkna roten
    Rot = sqrt ( Index );

    // skriv ut med två decimaler
    cout « setw ( 2 ) « Index « ": ";
    cout « setw ( 4 ) « setprecision ( 2 ) « Rot;

    // beräkna fjärderoten
    Rot = sqrt ( Rot );

    // skriv ut med fyra decimaler
    cout « ", " « setw ( 6 ) « setprecision ( 4 ) « Rot « endl;
  }
}
```

Körning av programmet ovan ger:

```
% ./SqareRoot2_2
 1: 1.00, 1.0000
 2: 1.41, 1.1892
 3: 1.73, 1.3161
 4: 2.00, 1.4142
 5: 2.24, 1.4953
 6: 2.45, 1.5651
 7: 2.65, 1.6266
 8: 2.83, 1.6818
 9: 3.00, 1.7321
10: 3.16, 1.7783
%
```

Här har även notationen satts till `ios::fixed`, eftersom vi då får nollor som följer decimalpunkten, och således snygga staplar med decimalkommat i samma kolumn. Nu

kan vi ordna utskrifter snyggt i staplar, men de är högerjusterade. Även detta kan ändras, och det görs igen med `setf()`. Det finns några konstanter inom området `ios::adjustfield` som heter:

- `ios::left` som skriver ut vänsterjusterad text.
- `ios::right` som skriver ut högerjusterad text.
- `ios::internal` som skriver ut ett eventuellt tecken vänsterjusterat och värdet högerjusterat.

Dessa kan användas t.ex. på följande sätt:

```
cout.setf ( ios::left, ios::adjustfield );
cout.setf ( ios::internal, ios::adjustfield );
```

8.3. Inmatning från tangentbordet

Förutom att endast skriva ut data till skärmen vill man förr eller senare kunna göra interaktiva program som tar input från användaren via tangentbordet. Även detta är lätt att göra i C++. För input finns en global variabel liknande `cout` som heter `cin`. Största skillnaden är att man tillsammans med `cin` använder sig av ». Man kan alltså se dem som "pilar" som anger åt vilket håll dataströmmen är riktad. Denna dataström brukar kallas *standard in*, i motsats till *standard out*.

Inmatning i C++ sköts direkt in i variabler, d.v.s « "pekar" på en variabel som skall tilldelas det inlästa värdet. Allmänt ser det ut så här:

```
cin » variabel;
cin » variabel1 » variabel2 » ... » variabelN;
```

Det är alltså möjligt att även läsa in flera värdet samtidigt. Värdena läses in från vänster till höger. Ett enkelt exempel som läser in ett tecken i taget tills ett visst tecken givits är:

```
char Tecken;
```

```
// upprepa slingan ända tills användaren ger in tecknet 'q'
do {
  // läs in ett tecken
  cout « "Ge ett tecken: ";
  cin » Tecken;
  cout « "Du gav in tecknet '" « Tecken « "'." « endl;
}
while ( Tecken != 'q' );
```

Om du provar programmet i verkligheten kommer du att märka att programmet inte reagerar trots att du skriver ett tecken. Inget händer fast flera tecken skrivs. Input i C++ är buffrat *radvis*, och en rad avslutas av att *enter* trycks. Efter detta skickas hela raden till programmet som sedan får en buffer som innehåller ett eller flera tecken som kan behandlas. Om du t.ex. ger in tre tecken och sedan trycker enter komer du att få tre rader utskrift. När programmet behandlat alla tecken som skrivits skriv en prompt ut på nytt och programmet väntar sig mera input. Notera dock att även varje gång som input fanns färdigt i buffern skrivs en prompt ut, men användaren ges aldrig ett tillfälle att mata in data. Ett exemepel på hur ovanstående program kan fås att läsa två tecken samtidigt:

```
char Tecken1, Tecken2;

// upprepa slingan ända tills användaren ger in tecknet 'q'
do {
  // läs in ett tecken
  cout « "Ge två tecken: ";
  cin » Tecken1 » Tecken2;
  cout « "Du gav in teck-
nen '" « Tecken1 « "' och '" « Tecken2 « "'." « endl;
}
while ( Tecken1 != 'q' && Tecken2 != 'q' );
```

Även här skrivs prompten ut ifall flera än två tecken givits. Programmet kommer att behandla två tecken åt gången, så om t.ex. tre tecken ges in första gången räcker det andra gången med att ge in ett enda tecken. Ges endast ett tecken första gånger händer ingenting utan programmet väntar sig ett tecken till följt av *enter*.

Förutom tecken kan man från `cin` även läsa in andra typer av data, t.ex. `int`, `string` och `float`. Ett enkelt exempel som läser in dessa datatyper ser ut så här:

```
int Tal;
float Flyttal;
string Text;

// läs ett tal
cout « "Ge ett heltal: " « flush;
cin » Tal;

// läs ett decimaltal
cout « "Ge ett decimaltal: " « flush;
cin » Flyttal;

// läs en sträng
cout « "Ge en sträng: " « flush;
cin » Text;

cout « "Du gav värdena '"
     « Tal « "', '"
     « Flyttal « "', '"
     « Text « "'." « endl;
```

Om man provar programmet och ger in ett illegalt värde på t.ex. heltalet kommer programmet att försöka tolka talet som ett heltal, men utan att lyckas. Följden blir för det mesta ett felaktigt värde och att de övriga värdena som läses in även får illegala värden. Strängen som läses in sist accepterar vilka värden som helst, eftersom en sträng inte försöker konvertera den lästa texten på något sätt. Om du försöker ge in en sträng som består av flera än ett enda ord kommer du att märka att strängen bryts vid det första mellanslaget eller tomrummet. Mellanslag används för att avsluta inläsning av en sträng. Du kommer att märka att det kan vara knepigt att göra ett program som läser input från användaren på ett robust sätt, speciellt om många olika datatyper används och programmet bör klara av att hantera olika typer av illegala värden. Det är inte omöjligt, men det är svårt att göra en robust lösning.

Notera även att C++ tillåter att decimaltal matas in enligt den "vetenskapliga" notationen, t.ex. följande tal är godtagbara flyttal:

```
1.05
-0.0003
9.8e5
-0.0001e-1
6.02e24
```

8.4. Utskrift till `cerr` och `clog`

Förutom `cout` och `cin` finns det även två andra fördefinierade globala strömmar. Dessa heter `cerr` och `clog`. Data som skrivs till dess strömmar går i normala fall till samma ställe som `cout`, d.v.s. skärmen. Skillnaden mellan `cout` och `cerr` är i hur de buffras. Ingen buffring förekommer då man använder `cerr`, så man kan tänka sig `cerr` som en normal `cout` där skriven data alltid följs av en `flush`. Normalt används `cerr` för utskrifter vid felsituationer, och då vill man kunna se meddelandena omedelbart utan buffring. Det går även att då programmet körs att styra `cout` och `cerr` till olika ställen, t.ex. `cerr` till en fil eller `/dev/null`, men detta är beroende på den shell (kommandotolk) som används. felsituationer

`clog` fungerar sedan som `cout`, och har således buffrar. I normalt bruk använder program `cout`, och ibland `cerr` för felmeddelanden.

Kapitel 9. Avancerade datatyper

Detta kapitel behandlar mera avancerade och sammansatta datatyper som används i C++. Pekare introduceras även.

9.1. Pekare

Vi har så här långt undvikit att använda en av de fundamentala byggstenarna i klassisk C (och C++), nämligen *pekare*. Då vi använt variabler tidigare har vi använt variablerna direkt, eller så kopior av dessa. Referensparametrar påminner till en viss mån om pekare (se Avsnitt 7.3), eftersom man med hjälp av dessa kan i en funktion ändra på värdet av en variabel så att det även "syns" i urprungsvariabeln. Med hjälp av pekare kan man åstadkomma samma funktionalitet. En pekare är en datatyp som innehåller *minnesadressen* för en variabel av någon typ. Med hjälp av denna adress kan man direkt manipulera variabeln.

Förståelse av pekare innebär att man måste förstå på vilket sätt data lagras i minnet, att varje variabel finns på en viss minnesadress, samt att variabelns värde kan manipuleras via denna minnesadress (om den är känd). För att få minnesadressen för en variabel skall man använda *adressoperatorn* `&`. Följande exempel skriver ut adresserna på två variabler:

```
#include <iostream>

int main () {
  int Tal;
  double AnnatTal;

  // skriv ut adressen på talen
  cout « "Tal har adressen     : " « &Tal « endl;
  cout « "AnnatTal har adressen: " « &AnnatTal « endl;
}
```

Programmet ger på den använda datorn ut nedanstående utskrift. Notera att dessa värden troligtvis ändrar mellan varje körning av programmet.

```
% Pointers1
Tal har adressen     : 0xbffff854
AnnatTal har adressen: 0xbffff848
%
```

Adresserna skrivs ut i hexadecimal bas. För att deklarera en pekare använder man denna allmänna form:

```
datatyp * variabelnamn;
```

Man definierar alltså pekare att peka på en viss typ av data. En asterisk `*` indikerar att det är frågan om en pekare.

> **Varning**
>
> Blanda inte ihop adressoperatorn `&` med den binära operatorn `&` eller `&` då det används för att indikera en referensparameter. Se även upp med `*` då det kan användas som både pekardefinition och multiplikationsoperator.

Exempel på pekardefinitioner är:

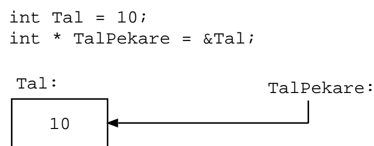
```
int * TalPekare1;
string * TextPekare;
float * FlyttalsP1, * FlyttalsP2;
```

för att tilldela en pekare adressen på en variabel använder man följande allmänna definition:

```
datatyp variabelnamn;
datatyp * pekarnamn;
pekarnamn = &variabelnamn;
```

Adressen som tilldelas en pekare måste vara adressen på en variabler av samma datatyp som pekaren är definierad att peka på. Det är således illegalt att tilldela en `float *` värdet på en `string`. Schematiskt ser det ut som nedan då en tilldelning gjorts:

Figur 9-1. Schematisk bild över pekare



Följande lilla exempel kan användas för att visa vilken adress en pekare pekar på:

```
#include <iostream>

int main () {

  double Tal;
  double * TalPekare = &Tal;

  // skriv ut adressen på talet och pekarens adress
  cout « "Tal har adressen       : " « &Tal « endl;
  cout « "TalPekare har adressen : " « TalPekare « endl;
}
```

På den använda maskinen ger programmet följande utskrift:

```
% Pointers2
Tal har adressen      : 0xbffff850
TalPekare har adressen: 0xbffff850
%
```

Man kan således enkelt skriva ut både pekare och minnesadresser. Notera att en pekare som inte initialiserats att peka på en viss minnesadress (variabel) har ett odefinierat värde. Om man inte direkt skall använda en pekare kan det löna sig att tilldela den

värdet 0 för att intialisera den till ett känt värde. Detta värde är sedan enkelt att känna igen i en debugger om någonting går fel. Initialisera t.ex. så här:

```
double Tal;
double * TalPekare = 0;
...
// långt senare tilldelas den ett värde
TalPekare = &Tal;
```

9.1.1. Avreferera pekare

Pekare är inte till mycken glädje om man endast kan använda dem till att skriva ut adresser. Som redan tidigare nämnts så kan man även accessera det data som en pekare refererar till. Detta görs genom att använda *avrefereringsoperatorn* `*`. Symbolen `*` har många användningsområden. Allmänt ser det ut så här:

```
*pekare;
```

Ett exempel klargör vad det är frågan om:

```
#include <iostream>

int main () {

  double Tal;
  double * TalPekare = &Tal;

  // skriv ut adressen på talet och pekarens adress
  cout « "Tal har värdet       : " « Tal « endl;
  cout « "*TalPekare har värdet: " « *TalPekare « endl;
}
```

På den använda maskinen ger programmet följande utskrift:

```
% Pointers3
Tal har värdet       : 10
*TalPekare har värdet: 10
%
```

Både `Tal` och `*TalPekare` gav samma värde. Så länge som pekaren pekar på `Tal` kommer det alltid att vara så. Man kan använda en *avrefererad* pekare i alla sammanhang som man kan använda variabeln som pekaren pekar på. Man kan säga att en avrefererad pekre har samma datatyp som datatypen den pekar på. I exemplet ovan skulle `Talpekare` kunna användas i alla sammanhang där `double` kan användas. Man kan t.ex. tilldela eller jämföra en avrefererad pekare.

> **Varning**
>
> Pekare i oförsiktiga händer är ett enkelt sätt att introducera allvarliga fel som kan vara svåra att hitta! En alltför stor del av programfel i C++ förorsakas av felaktig hantering av pekare.

9.1.2. Pekare som parametrar

Pekare kan förstås även skickas som parametrar till funktioner och även returneras som funktionsvärden. Genom att skicka pekare till en funktion kan den mottagande funktionen ändra på det värde pekaren pekar på. Samma effekt kan nås om man istället för pekare använder referensparametrar som specificeras med tecknet `&`. De båda påminner mycket om varandra, och största skillnaden är hur man refererar till dem. Pekare måste avrefereras med referensparametrar inte behöver göra det. Om man vill skicka parametrar som skall kunna modifieras till funktioner är det egalt vilket sätt som används. Ett exempel på en funktion som läser in ett värde i en funktion och skriver ut det i en annan:

```
#include <iostream>

void in (int * Tal) {
  // läs in ett tal
  cout « "Ge in ett tal: ";
  cin » *Tal;
}

void ut (int * Tal) {
```

```
  // skriv ut vårt tal
  cout « "Talet är: " « *Tal « endl;
}

int main () {

  int Tal;

  // läs in och skriv sedan ut
  in ( &Tal );
  ut ( &Tal );
}
```

Pekare kan alltså hanteras som vilken annan datatyp som helst då det gäller att skicka dem som parametrar. Notera att om en pekare skickas som parameter kan endast värdet på det som pekaren pekar på ändras, *inte* själva adressen den pekar på. I exemplet ovan kan t.ex. `in()` inte ändra `Tal` att peka på en annan variabel. Minnesadressen som skickats till `int()` och `ut()` kan inte ändras, den är "kopierad" från `main()`. Vill man kunna ändra den minnesadress en pekare pekar på i t.ex. `in()` måste en *pekare till en pekare* skickas som parameter. Detta skrivs med två `**`. Det är dock sällan man behöver anända pekare till pekare.

9.1.3. Konstanta pekare

Det går även definiera pekare som konstanter på olika sätt. Man kanske vill skicka en pekare till någon speciell datatyp som en parameter till en funktion, men inte vill att mottagaren skall kunna ändra det pekaren pekar på. I så fall kan man skapa konstanta pekare och använda `const` som extra parameterbeteckning i den mottagande funktionen. En konstant pekare kan deklareras t.ex. på följande sätt:

```
const int Tal = 100;
const int * KonstantPekare = &Tal;
```

En funktion som tar en konstant som parameter pekare kan se ut på följande sätt:

```
void berakna (const int * Tal) {
```

```
  ...
}
```

Man kan även definiera en pekare själv som konstant, trots att det den pekar på inte är det. I så fall kan man använda följande form:

```
int Tal = 100;
int * const KonstantPekare = &Tal;
```

I detta fall kan man nog ändra värdet på det som `KonstantPekare` pekar på, men däremot *inte* på pekaren själv. Den är alltså "låst" att peka på en och samma minnesadress. Med en `const int *` kan man nog ändra på själva pekarens värde, d.v.s. vilken minnesadress den pekar på, men däremot inte värdet på det den pekare på. En tredje form kombinerar dessa båda formerna och skapar en konstant pekare där man inte kan ändra på vare sig pekaren eller det pekaren pekar på:

```
int Tal = 100;
const int * const KonstantPekare = &Tal;
```

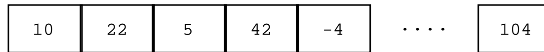
Speciellt den första formen med pekare som inte kan ändra på det de pekar på används ofta då man skickar data som funktionsparametrar. Numera verkar man dock även i de fallen gå över till att använda referensparametrar istället, och definiera parametrarna som `const` istället. Genom att använda referensparametrar behöver man inte komma ihåg vilka funktioner som skall ha pekare och vilka funktioner som skall ha normala parametrar. Använder man referensparametrar med `const` på lämpliga ställen är funktionerna lättare att använda ur anroparens synpunkt.

9.2. Vektorer

Vi skall nu behandla den första sammansatta datatypen i C++, nämligen *vektorer* (kallas även arrays). En vektor i C++ fungerar precis som i vilket programmeringsspråk som helst, men syntaxen för deklarering och användning skiljer sig förstås lite från andra språk. En vektor är en rad med element av en viss datatyp som finns tätt packade

sekventiellt i minnet. En vektor av `int` kan ses som på nedanstående sätt:

Figur 9-2. Schematisk bild över en `int`-vektor



En vektor innehåller alltid ett visst antal element, och vart och ett av dessa element är unikt och kan användas separat från de övriga. Det kan läsas, tilldelas, jämföras o.s.v. Allmänt deklareras en vektor i C++ på följande sätt:

```
datatyp vektornamn [storlek];
```

Man skall alltså ge en storlek på vektorn då den definieras. Några vektordefinitioner finns nedan:

```
int TalFoljd [16];
double Koordinater [3];
string Meningar [1024];
```

Den maximala storleken på vektorer är i teorin obegränsad, men i praktiken tar det tillgängliga minne slut i något skede. Storleken bör vara positiv. Storleken bör tolkas som antalet element vektorn kan innehålla. Då en vektor deklareras kan man även direkt initiera dess innehåll. Detta görs genom att lista elementen i ordningsföljd innanför { }. Detta kan endast göras då vektorn deklareras. Exempel:

```
int BitVarden[8] = { 1, 2, 4, 8, 16, 32, 64, 128 };
```

Vektorn `BitVarden` är nu en vektor med åtta element. Det är inte tillåtet att göra följande:

```
int BitVarden[8];
BitVarden = { 1, 2, 4, 8, 16, 32, 64, 128 };
```

Man kan däremot lämna bort storleken på vektorn och låta kompilatorn fylla i den om man vill. Ovanstående skulle då bli:

```
int BitVarden[] = { 1, 2, 4, 8, 16, 32, 64, 128 };
```

Antalet element i vektorn kan då räknas ut som *totala storleken / storleken på ett element*. Ovannämnda vektors storlek skulle t.ex. bli:

```
int BitVarden[] = { 1, 2, 4, 8, 16, 32, 64, 128 };
int Antal = sizeof (BitVarden) / sizeof (int);
```

9.2.1. Indexering av vektorer

För att en vektor skall kunna användas bör man kunna *indexera* de enskilda elementen i vektorn. Med indexering avses att de enskilda elementen kan manipuleras på olika sätt. I detta avseende använder C++ samma syntax som många andra språk. Allmänt ser det ut så här:

```
vektornamn [index];
```

Man ger alltså ett *index*, varvid vektorn ger det element som finns på den givna platsen i vektorn. Observera att vektorer alltid indexeras från 0 i C++! En vektor som är definierar som `int TalFoljd [10];` har 10 element, med index 0 till 9. Att indexera en vektor fel är ett vanligt fel man kan göra i C++. Vad händer om man indexerar fel? Då refererar man minne som inte tillhör vektorn, utan som kan tillhöra någon helt annan variabel som råkar finnas efter den avsedda vektorn. Resultatet är i de flesta fall inte det avsedda. Ett enkelt program som fyller en vektor med kvadrerade tal och skriver ut dem senare finns nedan:

```
#include <iostream>
#include <iomanip>

int main () {
  int Tal [10];

  // iterera och fyll vår vektor
```

```
  for ( int Index = 0; Index < 10; Index++ ) {
    Tal [Index] = Index * Index;
  }

  // iterera nu och skriv ut talen
  cout « "Tal  kvadrat" « endl;
  for ( int Index = 0; Index < 10; Index++ ) {
    cout « setw (3) « Index « setw (9) « Tal [Index] « endl;
  }
}
```

Precis som en avrefererad pekare kan en indexerad vektor användas på alla ställen där datatypen normalt kan användas. Programmet ovan ger följande utskrift:

```
% Vectors1
Tal  kvadrat
  0        0
  1        1
  2        4
  3        9
  4       16
  5       25
  6       36
  7       49
  8       64
  9       81
%
```

9.2.2. Tilldelning och jämförelse av vektorer

I C++ kan man inte jämföra vektorer med varandra direkt med operatorn == av en orsak som visas senare. Istället måste man kontrollera likhet genom att jämföra vektorernas innehåll, d.v.s man måste jämföra varje element i taget, och på så vis komma fram till om likhet gäller eller inte. Samma gäller för tilldelning. Man kan inte direkt tilldela en vektor värdet av en annan vektor, utan man måste tilldela elementen ett i taget. Detta förfarande kan verka en aning klumpigt.

9.3. Vektorer som pekare

Vektorer är en gammal datatyp som funnits med i standard C i långa tider. Därför är det inte förvånande att det går att manipulera vektorer även som pekare. I grund och botten är varje vektor en pekare till en minnesadress som har utrymme för ett visst antal element i följd av en viss datatyp. Om man lämnar bort `[]` på en vektor kan man hantera denna precis som en pekare. Exempelvis:

```
double TalVektor [8];
double * TalPekare;

TalPekare = TalVektor;
// eller
TalPekare = &TalVektor[0];
```

Således är en vektor alltid en pekare! I exemplet ovan är det senare sättet ekvivalent med det första, och med lite eftertanke märker man att det stämmer. `TalVektor` är en pekare och pekar på det första elementet i vektorn, d.v.s. `TalVektor[0]`, och tar man adressen för detta första element får man, just det, `TalPekare`.

9.3.1. Pekararitmetik

Eftersom vi nu har visat att varje vektor är en pekare till minnesadressen för första elementet i en följd, kan vi titta lite vidare på aritmetik med pekare. En pekare är ju i grund och botten ett heltal (`unsigned int`) som pekare på en minnesadress. Vi kan förstås manipulera detta tal med normala aritmetiska operationer, såsom `+` och `-`. Vi kan skriva om ovanstående vektorexempel så vi använder pekare istället:

```
#include <iostream>
#include <iomanip>

int main () {

  int Tal [10];

  // iterera och fyll vår vektor med hjälp av vektorn använd som
```

```
  // en pekare
  for ( int Index = 0; Index < 10; Index++ ) {
    *(Tal + Index) = Index * Index;
  }

  // deklarera en annan pekare till vektorn
  int * TalPekare = Tal;

  // iterera nu och skriv ut talen
  cout « "Tal  kvadrat" « endl;
  for ( int Index = 0; Index < 10; Index++ ) {
    cout « setw (3) « Index « setw (9) « *TalPekare++ « endl;
  }
}
```

Utskriften är samma som för föregående program. Exemplet kan dock behöva vissa förklaringar. Innan det en pekare pekar på kan användas måste den avrefereras. Det görs t.ex. på raden

```
*(Tal + Index) = Index * Index;
```

Här adderas ett tal till pekaren och sedan avrefereras denna. På så vis kan vi få pekaren att iterera över de olika positionerna i vektorn. Vi måste ha en parentes, eftersom avrefereringsoperatorn har samma precedens som `+`, så utan parentes skulle vi addera `Index` till det som `Tal` pekar på. En stor skillnad. Senare skapas en pekare `TalPekare` och sätts att peka på `Tal`:

```
int * TalPekare = Tal;
```

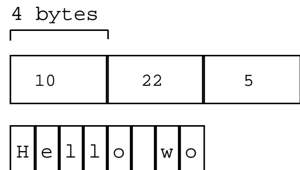
Denna pekare är inte nödvänding men vi får då en chans att visa hur vi kan ha en temporär pekare att iterera över elementen i en vektor med operatorn `++`:

```
*TalPekare++
```

Vad som händer här är att `++` fungerar som *postfixinkrementerande*, d.v.s. `TalPekare`:s värde ökas efter att den använts. Användingen i det här fallet är att den avrefereras. Efter användningen ökar `++` värdet på sin operand, alltså `TalPekare` med 1, och sätts således att peka på nästa element i vektorn.

Hur hänger det här ihop då? Den skarpsynte märker att en pekare är en normal `unsigned int` och operatorn `++` bode öka dess värde med 1, men det som `Tal` innehåller har troligtvis en storlek av 4 bytes per element. Operatorn `++` har alltså flyttat pekaren fram 4 minnesadresser!

Figur 9-3. Pekare och minnesadresser



I C++ är operatorn `++` så intelligent så den "vet" vilken datatyp en pekare pekar på, och kan således öka pekarens värde med så mycket som behövs för att flytta till nästa element. I fallet med t.ex. strägar som i bilden ovan flyttar `++` pekaren framåt så mycket som en `char` upptar, d.v.s. normalt 1 byte. Även `-`, `+=`, `-=` o.s.v. fungerar på samma sätt. I princip adderas (eller subtraheras) ett värde från pekaren som är lika stort som `sizeof (`*datatyp*`)`. Med hjälp av `++` och `-` kan man göra väldigt korta och effektiva iterationer över vektorer med hjälp av pekare. Ibland lönar det sig dock att offra lite effektivitet för att uppnå lättlästa program. Operatorn `[]` är trots allt lättare att förstå och läsa en direkt pekararitmetik! För den nyfikna kan nämnas att kompilatorer i de flesta fall översätter indexering med `[]` till pekararitmetik innan själva koden genereras.

9.3.2. Vektorer som funktionsparametrar

Vektorer kan förstås även skickas som parametrar till funktioner. Det finns två olika sätt att skicka vektorer som argument till funktioner, tack vare att vektorer är pekare. Man kan skicka dem som pekare eller som vektorer. Skickas de som pekare som vet mottagarfunktioner inte någonting om vektorns storlek, utan denna måste vara känd på något annat sätt, t.ex. via en annan parameter. Skickas den som en vektor så ges

storleken som en del av parameterdefinitionen. Exemplet nedan använder sig av de två olika sätten:

```
#include <iostream>

void in (int * Tal, int Antal) {
  // iterera och läs in tal
  for ( int Index = 0; Index < Antal; Index++ ) {
    cout « "Ge in tal " « Index + 1 « ": ";
    cin » *(Tal + Index);
  }
}

void ut (int Tal[3]) {
  // skriv ut vårt tal
  for ( int Index = 0; Index < 3; Index++ ) {
    cout « "Tal " « Index « " är: " « Tal[Index] « endl;
  }
}

int main () {
  int Tal [3];

  // läs in och skriv sedan ut
  in ( Tal, 3 );
  ut ( Tal );
}
```

Funktionen `in()` tar emot vektorn som en pekare. Enbart på bas av parameterdefinitionen vet man inte om det är frågan om en pekare till en enskild `int` eller om det är en pekare till första elementet i en vektor, därför skickas den extra parametern `Antal` med. Funktionen `ut()` däremot tar emot vektorn som en vektor, och då är ju storleken känd. Det senare sättet verkar ju vara mycket bättre, eftersom man då slipper bekymra sig om storlekar o.dyl., men den är även mera begränsad. Till `ut()` kan endast vektorer med tre element skickas, ingenting annat duger, medan `in()` är mycket mera flexibel och kan ta emot vilken storleks vektorer som helst. I båda fallen måste dock datatypen vara densamma hela tiden. Vilket sätt man förederar beror långt på till vad funktionen skall användas.

9.4. Flerdimensionella vektorer

Även i C++ kan man använda sig av *flerdimensionella vektorer*. Det är inget speciellt med dessa om man använt dylika i andra programmeringsspråk. Man använder dock sällan vektorer med fler än två dimensioner. Man deklarerar en flerdimensionell vektor allmänt på följande sätt:

```
datatyp variabelnamn [storlek1][storlek2]...[storlekN];
```

Exempel på en tvådimensionell vektor (matris) som fungerar som ett schackbräde:

```
int Chessboard[8][8];
```

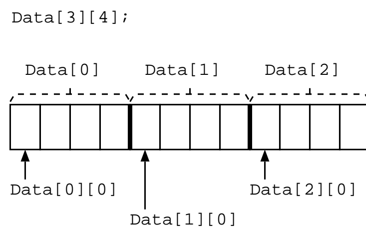
Indexering av flerdimensionella vektorer fungerar som för endimensionella vektorer, med den skillnaden att man skall ha ett index per dimension. Ett exempel på hur en tvådimensionell vektor kan fyllas med data och skrivas ut till skärmen:

```
#include <iostream>
#include <iomanip>

int main () {
  int Tal [8][4];

  // fyll matrisen med data
  for ( int Y = 0; Y < 8; Y++ ) {
    for ( int X = 0; X < 4; X++ ) {
      Tal [Y][X] = Y * 4 + X;
    }
  }

  // skriv ut matrisen till skärmen
  for ( int Y = 0; Y < 8; Y++ ) {
    for ( int X = 0; X < 4; X++ ) {
      cout « setw (3) « Tal [Y][X] « " ";
    }
    cout « endl;
  }
}
```

9.4.1. Flerdimensionella vektorer och minne

I minnet organiseras flerdimensionella vektorer sekventiellt precis som normala endimensionella vektorer. Om man t.ex. har en tvådimensionell vektor kan man se på den som en endimensionell vektor vars element är endimensionella vektorer. Följande figur illustrerar hur vektorns element är upplagda i minnet:

Figur 9-4. Schematisk bild över pekare



Vi ser att man i minnet kan se på `Data` som en serie på tre vektorer med fyra element.

Precis som med vanliga vektorer är flerdimensionella vektorer egentligen pekare till det första elementet. Man kan således även accessera flerdimensionella vektorer med hjälp av normal pekararitmetik (se Avsnitt 9.3.1).

9.5. Sammansatta datatyper

De datatyper som hittills behandlats har varit *primitiva* datatyper, d.v.s. bestått av endast en enda datatyp. Vektorer är en aning mera komplicerade, men de innehåller alltid en mängd av samma datatyp. I C++ kan man skapa *sammansatta* datatyper som är en sammansättning av flera olika datatyper. Varför vill man göra något sådant? Om vi t.ex. vill skapa ett litet enkelt adressregister som bokför personnamn, telefonnummer och kanske en adress, måste vi med våra kunskaper hittills skapa skilda vektorer med strängar för namn, adresser o.s.v. Det vore mera praktiskt att kunna gruppera all

information om en person i en enda variabel, och sedan gruppera dessa person-variabler i t.ex. en vektor. Här kommer sammansatta datatyper in. De tillåter programmeraren att skapa datatyper som är sammansättningar av de primitiva (och tidigare definierade sammansatta dattyperna), och på så vis bygga upp just den datatyp som behövs. Denna funktionalitet erbjuds av `struct` i C++. Allmänt ser en `struct` ut på följande sätt:

```
struct strukturnamn {
  datatyp1 variabelnamn1;
  datatyp2 variabelnamn2;
  ...
  datatypN variabelnamnN;
};
```

Definitionen börjar alltså med nyckelordet `struct`, följt av det namn som önskas på den nya sammansatta datatypen. Innanför { och `};` definieras sedan de variabler som datatypen skall innehålla. I vårt fall kunde vi definiera en datatyp `Person` på följande sätt:

```
struct Person {
  string Namn;
  string Adress;
  int    Telefonnummer;
};
```

Vår datatyp `Person` kan nu användas som vilken datatyp som helst överallt i vårt program där definitionen är känd. Notera att denna definition inte ännu definierat variabler av vår nya datatyp, detta måste göras separat:

```
Person Ingvar;
Person Elsa;
```

Variablerna `Ingvar` och `Elsa` kan användas som helt vanliga variabler, skickas som parametrar o.s.v.

> **Not:** I C++ behövs inget nyckelord `struct` då man deklarerar variabler av en sammansatt datatyp, vilket var obligatoriskt i C, där man måste definiera

variabeln `Ingvar` som `struct Person Ingvar;`. Detta har ändrats i C++, kanske närmast för att göra `struct` mera lik en klass (se Kapitel 14).

9.5.1. Scoping

Definitionen på en `struct` har även ett scope, d.v.s. det område i programmet där den är definierad. Man kan definiera en `struct` nästan var som helst i ett C++-program, och det ger vissa scoping-regler. En `struct` som är definierad inom en viss funktion kan endast användas inom den funktionen. Vill man att datatypen skall kunna användas av alla delar av ett program bör den placeras *externt*, d.v.s. utanför alla funktioner i början av programmet.

```
struct Kund {
  string Namn;
  string Adress;
};

void foobar () {
  // lokal definition, kan endast användas i denna funktion
  struct Rakning {
    string Vara;
    int Pris;
  }

  // definiera variabler av datatyper
  Kund K1;
  Rakning Spikar, Hammare;
  ...
}

int main () {
  // definiera variabler av datatyper
  Kund Foo, Bar;
  ...
}
```

I exemplet ovan innehåller funktionen `foobar` en egen lokal definition på en `struct`. Denna kan endast användas inom `foobar()`, eftersom dess scope inte sträcker sig till `main()`. Däremot kan `Kund` användas överallt i programmet, eftersom den är definierad utanför (och före) alla funktioner.

9.5.2. Accessera medlemmar

Vi har ingen glädje av `struct`:s om vi inte kan accessera datamedlemmar i en `struct`. Det är dock mycket enkelt att accessera och ändra data. Det görs genom att man avrefererar datamedlemmen med en punkt (`.`). För att t.ex. ge en ny telefonnummer åt medlemmen `Telefonnummer` i variabeln `Ingvar` gör man så här:

```
Person Ingvar;
Ingvar.Telefonnummer = 5556789;
```

Alla datamedlemmar kan acesseras med en punkt. Efter denna avreferering kan de användas precis på samma sätt som datatypen normalt kan användas. I exemplet ovan kan `Ingvar.Telefonnummer` användas precis som en normal `int`.

Då man skapar nya variabler av en `struct` kanske man vill direkt ge ett värde åt alla medlemmar. Då kan man göra enligt följande:

```
Person Ingvar = { "Ingvar Infåit", "DataCity", 5551234 };
```

Värdena sätts innanför `{` och `};` i exakt den ordning som de givits då datatypen `Person` definierats. Första värdet ges åt första medlemmen o.s.v. Vill man ge ett värde direkt är detta den enda möjligheten. Följande är *inte* tillåtet, och kommer att resultera i ett kompileringsfel:

```
Person Ingvar;
Ingvar = { "Ingvar Infåit", "DataCity", 5551234 };
```

9.5.3. Jämföra och tilldela

Att jämföra och tilldela `struct`:s kan vara lite knepigt. Tilldelning enligt följande är inte alltid möjlig:

```
Person Ingvar = { "Ingvar Infåit", "DataCity", 5551234 };
Person Elsa = Ingvar;
```

Det hela beror på innehållet i `struct`:en. Det som sker i en tilldelning som denna är att kompilatorn kommer att se till att `Elsa` tilldelas alla medlemmar från `Ingvar`. Ovanstående i princip ekvivalent med följande:

```
Person Ingvar = { "Ingvar Infåit", "DataCity", 5551234 };
Person Elsa;
Elsa.Namn = Ingvar.Namn;
Elsa.Adress = Ingvar.Adress;
Elsa.Telefonnummer = Ingvar.Telefonnummer;
```

För vårt exempel fungerar tilldelningen fint, eftersom tilldelningsoperatorn `=` är definierad för alla datatyper som används i `Person`. I alla sammanhang är det inte så, speciellt om olika främmande `struct`:s används. Om `=` inte är definierad får man i vissa fall resultat som inte är önskvärda. Man kan då explicit definiera en `=`-operator för `struct`:en, men i sådana fall är det i alla fall mera praktiskt att använda en klass (se Kapitel 14). Så länge som man endast använder primitiva datatyper i en `struct` kan man använda normal tilldelning.

Jämförelse av sammansatta dattyper är ännu svårare. Där kan man i normala fall aldrig direkt använda sig av operatorn `==` för att åstadkomma en jämförelse, utan man måste manuellt jämföra varje medlem. Detta kan vara ganska arbetsdrygt och fult om det är frågan om en stor datatyp. I kapitlet Kapitel 21 redogörs för hur man kan definiera operatorn `==` för klasser, men det samma kan även appliceras på `struct`:ar.

9.5.4. Sammansatta datyper som parametrar

Man kan givetvis skicka sammansatta datatyper som parametrar till funktioner. De fungerar precis som vilka datatyper som helst. Vill man kunna ändra på en `struct`

som skickats som parameter bör man skicka en pekare eller använda referensparametrar (`&`). Exemplet nedan läser in en `Produkt` och skriver ut denna:

```
#include <iostream>
#include <string>

struct Produkt {
  string Namn;
  float  Pris;
  int    Antal;
};

void nyProdukt (Produkt & P) {
  // läs in data i produktens medlemmar
  cout « "Namn : ";
  cin » P.Namn;
  cout « "Pris : ";
  cin » P.Pris;
  cout « "Antal: ";
  cin » P.Antal;
}

void skrivUt (Produkt * P) {
  // skriv ut info om produkten
  cout « endl;
  cout « "Information om produkt: " « P->Namn « endl;
  cout « "  pris:  " « (*P).Pris « endl;
  cout « "  antal: " « P->Antal « endl;
}

int main () {

  Produkt Ny;

  // läs in en produkt
  nyProdukt ( Ny );

  // skriv ut på skärmen
  skrivUt ( &Ny );
}
```

Körning av programmet kan t.ex. se ut på följande sätt:

```
% Struct4
Namn : C++-kompilator
Pris : 19.90
Antal: 3

Information om produkt:
  namn:  C++-kompilator
  pris:  19.9
  antal: 3
%
```

Här har vi nu använt oss av två sätt att skicka en `struct` som parameter. Det tredje sättet är det normala, d.v.s. helt som en normal variabel utan `&` eller pekare. Vi kunde använt det sättet för funktionen `skrivUt()`, eftersom den inte behöver ändra värden i `P`, men för `nyProdukt()` måste vi använda någondera sättet som tillåter oss att ändra värden i `P`. Notera att om man använder `&` kan man direkt referera till medlemmar med en `.`, men använder man pekare är det lite mera knepigt. Det lättare sättet då man använder pekare är att använda sig av *pilnotationen*. En `->` används för att avreferera en pekare som pekar på en `struct` (eller en klass) och accessera en medlem. Om man inte använder `->` bör man först avreferera pekaren med `*`, och därefter avreferera medlemmen med en punkt (`.`). Använder man `*`-metoden bör man notera att `.` har högre precedens än `*`, och evalueras således först, vilket ju inte är vad vi vill. Vi vill först evaluera `*` och därefter `.`. Vi måste således använda en parentes för att tvinga fram en viss precedensordning. Med tanke på detta är `->` att föredra. Notera att `->` endast kan användas tillsammans med pekare.

9.5.4.1. Effektivitet vid parameteröverföring

Hittills i våra program och exempel har vi inte behandlat effektivitet speciellt mycket. Då man hanterar sammansatta datatyper finns det en sak man bör notera, speciellt om man skickar dylika som parametrar till olika funktioner. Om man skickar en `struct` som en vanlig parameter (ej pekare och utan `&`) så kommer kompilatorn att kopiera hela innehållet i `struct`:en till mottagarfunktionen. I våra små exempel är det inte mycket som behöver kopieras, men i ett verkligt system kan man ha tiotals medelemmar i

dylika datatyper. Om varje funktionsanrop innebär att stora mängder data måste kopieras kan ett program enkelt bli långsamt. Lösningen till detta problem är att undvika att kopiera så mycket data! Man kan åstadkomma detta genom att t.ex. använda pekare eller referensparametrar. I båda fallen skickas endast adressen på variablen i fråga, och stora inbesparningar kan uppnås. Om `&` används kan mottagaren använda parametern precis på samma sätt som om den vore skickad på normalt sätt. Således rekommenderas att `&` används för överföring av sammansatta datatyper.

Gäller samma resonemang t.ex. vektorer, som ju kan vara väldigt stora? Nej, det gör det inte, eftersom kompilatorn alltid i kompileringsskedet ersätter vektor-parametrar med pekare. Därmed skickas alltid endast adressen på första elementet till den mottagande funktionen.

9.6. Enumereringar

Ibland kanske man vill använda någon form av namngivna konstanter för att representera något, t.ex. färger. Vi kan då t.ex. deklarera ett antal variabler av typen `int` och ge dem unika värden. Sedan kan vi använda dessa variabler för att t.ex. indexera en vektor med färger. Låter klumpigt, eller hur? Det finns ett lättare sätt att åstadkomma följder av tal i C++, nämligen med *enumereringar*. Allmänt definieras en enumerering på följande sätt:

```
enum namn { element1, ..., elementN };
```

En enumerering innehåller en mängd element, som kommer att enumerera börjande från 0. På så vis associeras varje element med ett unikt värde. En enumerering ges även ett namn, så att man kan skapa variabler med enumereringen som datatyp. Ett exempel på en enumererad regnbåge:

```
enum Color { Black, White, Red, Orange, Yellow, Green, Blue, Pur-
ple, Indigo };
```

På engelska för omväxlings skull. Här har vi nu definierat en ny datatyp `Colors`, och vi kan deklarera variabler av denna typ om vi vill:

```
Color EnFarg;
Color Regnbage [5];
```

Denna datatyp är dock speciell, i och med att variabler av en enumerad datatyp endast kan innehålla de värden som gavs vid enumereringens definition. Så i vårt exempel kan `EnFarg` ges alla värden från `Black` till `Indigo`. En variabel tilldelas ett värde helt normalt:

```
Farg = Red;
Regnbage [0] = Red;
Regnbage [1] = Orange;
```

De enda operationer som är tillåtna med enumereringar är tilldelning och jämförelse. Man kan t.ex. inte addera eller subtrahera enumereringar. Man kan även endast tilldela enumererade variabler värden som finns i enumereringen. Följande är alltså inte tillåtet:

```
Color Farg;
Farg = 10;    // fel datatyp
Farg = Gray;  // ingen sådan färg i enumereringen
```

Man kan tilldela en enumerering en `int` om man först konverterar den till en enumerering:

```
Color Farg;
Farg = (Color)3;  // ok, blir 'Orange'
Farg = (Color)8;  // ok, blir 'Indigo'
Farg = (Color)45; // fel, ingen sådan färg
```

Det sista exemplet är illegalt, eftersom det inte finns så många värden i enumereringen. Som redan tidigare nämnts så är en enumerering i grund och botten en `int`, därför är ovanstående konvertering möjlig. Man kan ge enumereringar andra värden än sådana som börjar från 0 och ökar med 1 per värde. Det görs genom att skriva värdet efter enumereringselementet. Om man t.ex. vill börja en serie från 10 kan man göra följande:

```
enum Bilar { Opel = 10, BMW, Mercedes, Ford };
```

Nu får `BMW` värdet 11, `Mercedes` värdet 12 och `Ford` värdet 13. Man kan även ge varje element ett unikt värde, men de bör vara i stigande ordning när man går högerut. Man kan även ge flera element samma värde om det finns behov för detta.

```
enum KursBitar { Forelasning = 1, Tent = 2, RO = 4, Kom-
pendium = 8, Studerande = 16 };
enum Error { Critical = 0, Fatal = 0, Serious, Warning, Ok };
```

Ovan har vi skapat en *binär* enumerering vars värden kan kombineras med t.ex. `or` för att på så vis kombinera flera flaggor i ett enda `int`-värde. Användingsområdena för enumereringar är mycket varierande. Ofta används de där man vill ha en egen datatyp som kan användas som en `int` (t.ex. för att indexera vektorer), eller där man vill ha en datatyp som endast kan ha ett fåtal fördefinierade värden.

9.7. Definiera egna datatyper

I C++ kan man definiera helt egna datatyper om man vill. Detta görs ju redan implicit då man skapar sammansatta datatyper eller använder enumereringar. Förutom dessa kan man definiera egna typer som har en existerande datatyp som grund. Detta görs med hjälp av nyckelordet `typedef`. Allmänna formen för en typdefinition är:

```
typedef ursprungsdatatyp ny_datatyp;
```

Man ger i princip en existerande datatyp ett nytt namn. Den nya datatypen fungerar exakt som den gamla typen med avseende på aritmetik, tilldelningar o.s.v. Några exempel på nya definierade typer:

```
typedef long ErrorCode;
typedef unsigned char uchar;
```

Via den första definitionen har vi en ny datatyp som heter `ErrorCode` som fungerar precis som en `int`. Den kan göra ett program lättare att läsa då man enkelt förstår vad en funktion returnerar om den returnerar en `ErrorCode`. Det andra fallet definierar en ny typ `uchar` som en ersättare för `unsigned char`. Det är kortare att skriva `uchar`, och om man använder dylika parametrar på många ställen kanske det är mera

överskådligt att använda sig av `uchar`. Här märker vi att endast det sista ordet i en `typedef` blir den nya datatypen, medan allt mellan det ordet och `typedef` anses tillhöra namnet på originaltypen. Det är alltså tillåtet att ha en ursprungsdatatyp med flera än ett ord i sin definition, t.ex. en `struct`-definition.

I de flesta fall kan man använda en enumerering eller en klass istället för `typedef`, så det är ganska sällan den används i praktiken. I vissa fall försvarar den dock sin plats i C++:s vokabulär.

Kapitel 10. Preprocessorn

Detta kapitel redogör för vad preprocessorn är och hur den fungerar i C++.

10.1. Vad är en preprocessor?

En preprocessor i C++ är ett program som på något sätt manipulerar programkoden *innan* den egentliga kompileringen utförs. I C++ finns det en preprocessor som körs automatiskt varje gång man kompilerar ett program. Den körs helt transparent av kompilatorn och man behöver egentligen inte veta om att den existerar. Preprocessorn har några viktiga funktioner:

- *inkludera filer*
- *hantera konstanter*
- *hantera makron*
- *skydda mot multipel inkludering*

Nedan behandlas de olika huvuduppgifterna separat.

10.2. Inkludera filer

Den viktigaste uppgiften som preprocessorn utför är *inkludering av filer*. Inkludering av filer sköts med hjälp av `#include`. Om man tittar på en lista av nyckelord som används i C++ finns ordet `include` inte med. Orsaken är enkel: C++-kompilatorn ser aldrig ordet `include` i den text som den kompilerar, preprocessorn har redan ersatt `#include`-definitionen med innehållet i korrekt fil.

Man använder alltså `#include` för att inkludera definitioner från andra filer. Hittills har vi använt `#include` för att inkludera information om bl.a. *streams* (`cin` och `cout`) och strängar. Vad vi egentligen gjort är att vi instruerat preprocessorn att inkludera

stora mängder definitioner för olika datatyper och funktioner. Vi kan sedan i vårt program använda alla de definitioner som finns i de inkluderade filerna. Allmänt ser en `include`-deklaration ut så här:

```
#include "filnamn"
#include <filnamn>
```

Notera att man inte behöver något semikolon (`;`) efter definitionen, eftersom det inte är frågan om en normal C++-instruktion. De två olika formerna av `#include` fungerar på olika sätt. Skillnaden ligger i var preprocessorn försöker hitta filerna. Den första definitionen kommer att försöka hitta *filnamn* i den aktuella arbetskatalogen först, och om filen inte hittades där, så genomsöks andra kataloger som antigen givits till kompilatorn eller som är kompilatorspecifika. Den andra definitionen gör att preprocessorn inte ens försöker hitta filen i aktuell arbetskatalog, utan direkt söker i de systemspecifika katalogerna (t.ex. under `/usr/include` på Unix). Den första formen används normalt om man har ett eget program som använder en definitionsfil. Några exempel på `#include`-definitioner:

```
#include <iostream>
#include <string>
#include <stdio.h>
#include "MinaDefinitioner.h"
#include "Konstanter"
```

Man brukar placera alla `#include`-deklarationer i början av programmen, eftersom inga funktioner som definieras i en header-fil kan användas innan de definierats, d.v.s. innan de inkluderats. Traditionellt har man gett header-filer extensionen `.h`, men i och med nya standarder för C++ är detta inte längre nödvändigt. Därför har många av de `#include`-definitioner vi använder ingen `.h`-extension. Mera information om header-filer och externa bibliotek finns i Kapitel 11.

10.2.1. Standarden för C++

Standarden för C++ som relativt nyligen har fått en relativt stabil form ändrar på hur header-filer skall namnges. Enligt denna standard skall *alla* header-filer vara utan

extensionen `.h`. Vi har ju redan sett att alla C++-specifika header-filer redan saknar denna extension, men gamla C-specifika header-filer ännu använder sig av extensionen. Dessa skall istället får ett prefix `c` och sedan även lämna bort extensionen `.h`. Vi får då att t.ex. `stdio.h` och `string.h` blir enligt standarden `cstdio` och `cstring`.

Det kan dock ta en tid innan alla kompilatorer fullt gått in för denna namngivning av header-filer, främst för att det finns enorma mängder gamla C-program som är beroende av den gamla namngivningen. På grund av detta kommer de två sätten att finnas parallellt ännu en lång tid, kanske för alltid. Det rekommenderas dock att man använder sig av den nya namngivningen om kompilatorn understöder detta, även om programmet i detta kompendium inte gör det.

10.3. Konstanter

I traditionell C har man använt preprocessorn för att definiera olika typer av konstanter. I C++ gör man hellre dylika definitioner med `const`, men det gamla sättet finns kvar för dem som vill använda det. Man använder här preprocessorns nyckelord `#define`. Notera att inte heller detta ord är ett reserverat ord i C++ (eller C). Den allmänna formen för en konstantdefinition är:

```
#define konstantnamn värde
```

Här behöver man inte heller använda ett semikolon för att avsluta raden, eftersom det är frågan om ett preprocessordirektiv som aldrig ens ses av kompilatorn. Några exempel på definitioner är.

```
#define MEDDELANDE "Hello World!"
#define MIN_VARDE  0
#define MaxVarde   100
```

Traditionellt har definitioner till preprocessorn alltid getts namn med stora bokstäver, närmast för att separera dessa från normala variabler. Konstanter som görs med `const` i C++ skrivs däremot normalt med både små och stora bokstäver. Det som preprocessorn gör då den stöter på t.ex. `MEDDELANDE` ovan i en programfil är att den helt enkelt

ersätter texten `MEDDELANDE` med texten `"Hello World!"`, alltså normal textsubstitution. För kompilatorn ser texten sedan ut som vilken text eller vilket tal som helst. Man kan avdefiniera en konstant genom att använda `#undef`. Vi kan t.ex. avdefiniera `MaxVarde` och ersätta den med en med stora bokstäver:

```
#undef MaxVarde
#define MAX_VARDE 100
```

Vi kommer nedan att se ett sammanhang där `#define` används i stor grad ännu idag. I övriga sammanhang rekommenderas som sagt att konstanter definierade med `const` används istället.

10.4. Makron

Man kan även använda preprocessorn för att skapa *makron* av olika slag. Även denna funktionalitet brukar numera ersättas av en nyare konstruktion i C++. För att skapa makron använder man sig av `#define`, precis som då man skapar konstanter. En makro är egentligen en liten funktion som man kan ge argument åt. Skillnaden mellan ett makro och en vanlig funktion är att koden för ett makro placeras direkt på platsen för anropet, d.v.s. det blir inget funktionsanrop överhuvudtaget. Detta är en aning snabbare än att ha en separat funktion, men i längden även mera utrymmeskrävande, eftersom samma makro kan vara utspritt på många ställen i ett program. Ett makro som beräknar kvadraten på det givna argumentet kan se ut så här:

```
#define KVADRAT(X) X * X
```

Ett program kan sedan använda denna definition på följande sätt:

```
#include <iostream>

#define KVADRAT(X) X * X

int main () {
  cout « "Kvadraten på 5 = " « KVADRAT(5) « endl;
}
```

Vi har nu definierat en makro `KVADRAT` som tar ett argument. Texten `X * X` är den text som `KVADRAT` ersätts med, förstås med X ersatt med det värde som man skrivit in. Kompilatorn ser egentligen i vårt fall följande rad:

```
cout « "Kvadraten på 5 = " « 5 * 5 « endl;
```

Man bör vara försiktig då man definierar makron. T.ex. följande skulle inte fungera med vår makrodefinition:

```
cout « "Kvadraten på 2 + 3 = " « KVADRAT(2 + 3) « endl;
```

Det skulle bli följande text:

```
cout « "Kvadraten på 2 + 3 = " « 2 + 3 * 2 + 3 « endl;
```

vilket inte är vad vi avsåg. Vi kan förbättra vår makrodefinition med hjälp av parenteser för att tvinga fram den önskade evalueringsordningen:

```
#define KVADRAT2(X) ((X) * (X))
```

Denna definition fungerar för diverse aritmetiska uttryck. Man kan även skapa makron som tar flera än ett argument. Ett klassiskt exempel är ett makro som ger ut det större av två värden:

```
#define MAX(X,Y) (X > Y ? X : Y)
```

Även här använder vi parenteser för att bättre gruppera evalueringen. Vi använder oss av `?`-operatorn för att utföra jämförelsen. Med C++ kan man skriva betydligt mera flexibla makro-liknande funktioner med s.k. *inline*-kod. Dylik kod placeras även av kompilatorn direkt på anropets plats, så inget funktionsanrop görs.

10.4.1. Nackdelar med makron

Det finns alltså några ganska allvarliga begränsingar och nackdelar med att använda makron. Man bör noga överväga ifall fördelarna överväger nackdelarna. I de flesta fall

är det enklare att definiera en likadan funktion (eller metod) och låta kompliatorn sköta optimieringen av koden.

De olika nackdelarna är:

- *kräver mera minne* eftersom samma kod finns på många ställen i programmet.
- *svåra att konstruera rätt* så att man får parenteser o.dyl. på de ställen där de behövs. Vanligen märker man dock inte dessa ställen förrän man spenderat tid med att söka efter logiska fel i programmet.
- *ingen typchecking* vilket betyder att man kan t.ex. skicka en sträng till ett makro som utför en aritmetisk operation. Detta godkänner förstås inte kompilatorn, men det kan vara svårt att veta exakt vad som gått fel.

10.5. Multipel inkludering

En C++-kompilator är i princip ett ganska dumt program. Den kan endast använda definitioner som finns "tidigare" under kompileringen, så t.ex. funktioner måste vara definierade antingen helt eller via prototyper innan de används. Kompilatorn klarar inte heller av att en symbol (t.ex. funktion) är definierad två gånger, även om de har samma definition. Egentligen kunde kompilatorn vägra att kompilera nedanstående program eftersom definitionerna från `iostream` är inkluderade två gånger:

```
#include <iostream>
#include <iostream>

int main () {
  cout « "Hello world!" « endl;
}
```

Så är dock inte fallet. Det vore enormt svårt att planera hur filer skall inkludera varandra om samma symbol inte får förekomma två gånger i samma fil. Det finns ett "preprocessor-trick" man kan använda som löser problemet och gör att vi inte behöver

bry oss om problematiken överhuvudtaget. I t.ex. filen `iostream` finns följande definitioner (eller liknande):

```
// This file is part of the GNU ANSI C++ Library.

#ifndef __IOSTREAM__
#define __IOSTREAM__
// diverse definitioner
#endif
```

Här har vi två nya preprocessor-nyckelord, nämligen `#ifdef` och `#endif`. Om man studerar de preprocessordirektiv som förekommer i filen ovan kan man komma till slutsatsen att det är frågan om en rudimentär `if`-sats, men som används av preprocessorn. Det första ordet `#ifndef` skall läsas *if not defined*. Den kontrollerar om symbolen som följer efter är definierad, i detta fall då `__IOSTREAM__`. Om så inte är fallet fortsättet "evalueringen" in i `#ifndef`-satsen och direkt på nästa rad definierar symbolen `__IOSTREAM__`. Efter detta följer en hel del definitioner som inte får förekomma två gånger. Sist kommer en `#endif` som ju avslutar `if`-satsen. Om samma fil nu läses in en gång till under kompileringsskedet kommer `#ifndef` att vara falsk, eftersom symbolen `__IOSTREAM__` redan är definierad, och definitionerna innanför `if`-satsen omdefinieras inte. Detta system används av så gott som alla header-filer i C++. Systemet skyddar inte mot att *filen* läses multipla gånger, endast emot att symboler definieras multipla gånger.

Normalt definierar man en symbol som har anknytning till header-filens namn. Varje headerfil bör ha en symbol som är unik. Om man har samma symbol i flera filer kommer senare filer som använder samma symbol inte att läsas in korrekt. Om man således har t.ex. en fil som heter `Definitioner.h` bör man sätta någon form av inkluderingsskdd i denna, t.ex. på detta sätt:

```
#ifndef DEFINITIONER_H
#define DEFINITIONER_H

// diverse definitioner

#endif
```

Ingen annan fil bör använda samma symbol `DEFINITIONER_H`.

Notera att `DEFINITIONER_H` inte ges något värde, utan endast status som definierad. Det finns även ett preprocessordirektiv som kollar om en symbol redan är definierad som heter `#ifdef`, och läses *if defined*. Det finns även en `else`-del som kan användas om man vill kunna göra t.ex. två olika definitioner på av av något värde. Den skrivs `#else`. Ett användningsområde för `#ifdef` är för att skriva ut debuggnings-text på skärmen. T.ex. följande är en ganska normal konstruktion:

```
int main () {
  float X, Y, Z;

  // beräkna koordinater för X, Y och Z via någon formel
  X = ...
  Y = ...
  Z = ...

#ifdef DEBUG
  cout « "X=" « X « ", "Y=" « Y « ", Z=" « Z « endl;
#endif

  ...
}
```

Ovan används `#ifdef` för att kontrollera om en symbol `DEBUG` är definierad. Om den är definierad är programmet under debuggning och värdena för `X`, `Y` och `Z` skrivs ut på skärmen. I den slutgiltiga versionen av programmet skall ingen dylik text skrivas ut. Man kan kontrollera huruvida debuggning skall skrivas ut i detta fall genom att placera en `#define DEBUG` eller `#undef DEBUG` i någon gemensam headerfil.

Kapitel 11. Externa bibliotek

Detta kapitel redogör för hur externa bibliotek med fördefinierade funktioner fungerar.

11.1. Vad är bibliotek?

Ett *bibliotek* (library) i programmeringssammanhang är inte exakt det samma som ett bibliotek i normala sammanhang. I bibliotek samlas olika fördefinierade funktioner och symboler som andra program kan använda. Det finns normalt ett stort antal olika bibliotek som innehåller funktioner för olika behov. Endel av dessa bibliotek följer med operativsystemet, medan andra följer med kompilatorn eller externa programpaket. Funktioner i dessa bibliotek kan sedan användas av alla program.

11.1.1. Bibliotek under Unix

Under Unix finns olika bibliotek i katalogerna `/lib` och `/usr/lib`. Även andra platser används, men deras placerinb beror för det mesta på vilket system som används. Man kan titta under dessa kataloger och söka efter filer varsn namn börjar med texten `lib` och slutar på `.so`. I vissa fall förekommer även en eller flera siffror före eller efter `.so` för att indikera en versionsnummer. Några exempel på bibliotek på den maskin där detta skrivs är:

```
/lib/libc-2.1.1.so
/usr/lib/libMesaGL.so.3.1
```

Det första av dessa bibliotek innehåller normala funktioner som nästan varje program använder sig av, medan det senare är ett bibliotek som innehåller funktioner för *OpenGL*.

11.1.2. Bibliotek under Windows

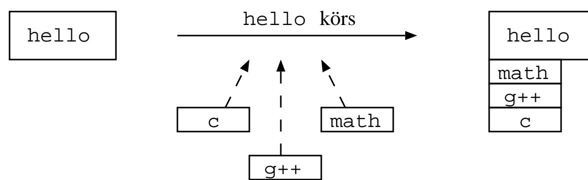
Bibliotek under Windows fungerar i princip på samma sätt som under Unix. Där har bibliotek normalt extensionen `.dll`, som står för *dynamic linking library*. Placeringen av dessa bibliotek varierar mellan versioner av Windows och till vilken applikation biblioteket hör.

11.2. Hur bibliotek fungerar

Bibliotek används både då program kompileras och då de körs. Man måste berätta åt kompilatorn från vilka bibliotek man har använt funktioner och kompilatorn sköter sedan om att dessa bibliotek används av programmet då det körs. Vid kompileringsskedet kontrolleras alla funktioner som kommer från något externt bibliotek och kod genereras för att kunna anropa dessa funktioner då programmet körs. Man brukar säga att bilioteket *dynamiskt länkas* in i programmet. Då programmet sedan körs laddas först själv programmet in, därefter bibliotek som programmet använder. Om andra program redan använder samma bibliotek finns de redan färdigt inladdade i minnet. Man talar om att ett bibliotek är *delat* (shared). Alla program som kör samtidigt delar samma instans av ett bibliotek, så det finns inläst endast en gång i minnet. Detta leder till mera minnessnåla program, eftersom en stor del av koden kan delas mellan olika program som kör samtidigt.

Så gott som alla program som verkligen gör något använder sig av ett eller flera bibliotek. Ibland använder ett bibliotek ett annat bibliotek, så ett program kan, utan att veta det, använda ett stort antal olika bibliotek. I de flesta fall behöver man egentligen inte veta detta, utan endast då man explicit vill använda funktioner ur ett extra bibliotek, t.ex. då man använder OpenGL, måste man berätta åt kompilatorn vilket bibliotek som skall användas.

Figur 11-1. Länkning av bibliotek vid körning



Bilden ovan illustrerar hur exempel-biblioteken `c`, `g++` och `math` länkas in då `hello` körs. De fungerar sedan som en del av programmet under den tid det körs.

11.2.1. Bibliotek och header-filer

För att programmerare skall veta vilka funktioner som finns i olika bibliotek har man inom C++-världen traditionellt använt någonting som kallas *header-filer*. Dessa är filer som innehåller olika definitioner, funktionsprototyper m.m., och som kan användas av program. En header-fil berättar vad som finns i biblioteket och vad som kan användas. Utan header-filer (och dokumentation för biblioteken) är det mycket svårt att veta vad som finns i ett bibliotek. Ett program kan uttrycka en önskan att använda funktioner från ett bibliotek genom att ha en eller flera `#include`-deklarationer någonstans i programmet. Ofta finns det flera olika header-filer för ett bibliotek, men ett program kanske endast behöver inkludera en av dessa, beroende på vilka funktioner eller definitioner som önskas.

De program vi hittills skapat har i de flesta fall inkluderat headerfilen `iostream`. Denna headerfil är en C++-headerfil och innehåller de flesta definitioner vi har behövt för att hantera I/O. Vissa funktioner har inte funnits i denna header-fil, så då har även `iomanip` inkluderats. Båda dessa header-filer definierar funktioner som finns i bibliotek som automatiskt används (se Avsnitt 11.4).

11.3. Fördelen med bibliotek

Vad finns det då för fördelar med att placera en massa funktioner och definitioner i olika bibliotek, och hur fungerar det hela? Vem blir gladare av att det finns en hel del kod kringspridd i olika filer på olika ställen på datorn? Det finns några direkta fördelar med att ha *gemensam* kod i bibliotek.

- *mindre program* eftersom en stor del av programkoden finns i externa bibliotek. Programmen tar då mindre utrymme på t.ex. disk.
- *kortare starttider* som en följd av att programmet är mindre och biblioteken som används ofta redan är inlästa av något annat program.
- *snabbare utvecklingstid* eftersom man kan använda färdiga funktioner som finns i bibliotek och som utvecklats av andra programmerare. Man behöver inte uppfinna hjulet på nytt varje gång.
- *robustare program* då kod i bibliotek ofta är noggrannare testad än kod man själv skulle skapa. Ofta är bibliotek resultat av åratal av programmeringsarbete.
- *snabbare program* eftersom olika bibliotek ofta är mycket väl optimerade. Detta är även ett resultat av åratal av många skickliga programmeras arbete.
- *programmering på högre nivå* så man inte behöver implementera grundläggande funktioner på operativsystemnivå. Biblioteken innehåller abstraktioner för olika lågnivå-resurser.
- *minnessnålare program* eftersom eftersom olika program använder sog av samma bibliotek som är inladdat i minnet. Ofta är över 75% av den mängd minne ett program upptar då det körs olika bibliotek som kan delas med andra program.

11.4. Exempelbibliotek

Vi har redan använt ett eller flera bibliotek i alla våra program. Bl.a. det normala C-biblioteket samt biblioteket som innehåller diverse C++-funktionalitet. Dessa används totalt automatiskt av kompilatorn vid kompileringsskedet och operativsystemet

vid körningskedet. Det finns andra bibliotek som innehåller olika funktionalitet man kan vara intresserad av att använda.

11.4.1. Matematikbibliotek

Många program behöver i något skede funktioner som t.ex. `sin()`, `cos()` eller `sqrt()` (kvadratrot). Dessa finns normalt i biblioteket `math`. Detta är vanligen det första externa bibliotek man stöter på då man börjat programmera C eller C++. Funktionerna i detta bibliotek är i de flesta fall definierade i header-filen `math.h`. För att kompilatorn under Unix skall använda detta bibliotek måste man ge en explicit parameter som säger att matematikbiblioteket skall användas. Denna parameter är `-lm`. I grafiska programmeringsmiljöer brukar detta bibliotek länkas in automatiskt om det behövs. Se mera info i Avsnitt 13.2.

11.4.2. X11 och Windows

Både X11 och Windows har ett antal olika bibliotek som tillhandahåller funktioner så att program kan bygga *grafiska gränssnitt*. Under Unix finns det ett antal olika bibliotek eller biblioteksamlingar såsom t.ex. KDE, Qt och Gtk. Båda dessa bibliotek bygger i sin tur på biblioteket X11, som innehåller lågnivåfunktioner för hantering av GUI:s (graphical user interface). Användingen av detta bibliotek faller utanför denna kurs omfång, men det finns rikligt med material på nätet.

11.4.3. OpenGL

OpenGL är ett populärt bibliotek under både Unix och Windows för att skapa 3D-grafik. Ett stort antal funktioner finns för att skapa olika former av 3D-modeller för t.ex. visualisering, spel eller användargränssnitt. Den version som används under t.ex. Linux heter dock *Mesa*, men innehåller exakt samma funktioner som den normala OpenGL. Användingen av detta biblotek är även utanför denna kurs omfång. Även här finns rikligt med material på nätet, både för nybörjare och avancerade programmerare.

Kapitel 12. Dynamisk minneshantering

Detta kapitel innehåller information om hur man dynamiskt kan allokera och frigöra minne i C++. På så vis kan man skapa program som kan använda minne enligt behov.

12.1. Vad är dynamisk minneshantering

De program vi hittills tittat på har inte använt någon form av dynamisk minneshantering. Allt minne som använts har varit deklarerat som antingen vektorer eller vanliga variabler. Denna modell fungerar fint om man vid programmeringsskedet vet *exakt* hur mycket data som kommer att behövas av programmet under hela dess körningstid. I normala fall vet man inte det, speciellt om programmet är mera komplicerat än de enkla exempel som vi hittills tittat på. En brutal lösning är att i programmet skapa några överdrivet stora vektorer som kan hålla det data som behövs, men denna lösning har flera brister:

- programmet slösar med minne. Om det innehåller ett antal stora vektorer som ändå för det mesta är tomma slösas mycket minne.
- programmet kanske ändå någongång behöver mera minne än vad som finns i de statiska vektorerna.

Alldeles tydligt behövs det någon form av funktionalitet för ett program att allokera mera minne under programmets körning, exakt då minnet behövs, och sedan funktionalitet för att ge obehövligt minne tillbaka till operativsystemet. Detta är vad dynamisk minnesallokering handlar om.

12.2. Allokering och frigöring av minne

För att kunna allokera och frigöra minne dynamiskt måste man i C++ ha något "ställe" dit minnet kan placeras då det allokerats. I C++ används pekare för att referera till

allokerat minne. Via pekaren kan minnet sedan avefereras och användas.

12.2.1. Allokering av minne

Allokering av minne sker med operatorn `new`. Allmänt ser en allokering ut på följande sätt:

```
datatyp * pekarnamn = new datatyp;
```

Man bör alltså ha en pekare till den datatypen man ämnar allokera minne för, t.ex. `int`. Operatorn `new` påminner om en funktion, men det är egentligen en operator, så den behöver inte parenteser runt sin parameter. Datatypen som ges till `new` måste vara av samma typ som den pekare som skall ta emot det minne som `new` returnerar. Efter en allokering kan man avreferera den använda pekaren och använda denna som vilken variabel som helst. Några exempel på allokeringar:

```
int * IntPekare = new int;
float * FloatPekare;
FloatPekare = new float;
string * Text = new string;
```

Man behöver alltså inte direkt allokera minne till en pekare, utan man kan först deklarera en pekare och sedan allokera minne i ett senare skede. Man kan även allokera minne flera gånger med hjälp av samma pekare. Flera pekare kan även peka på samma minnesområde, precis som flera pekare kan peka på samma variabels minnesadress. Man bör dock observera att om det finns endast en pekare till ett allokerat minnesblock, och denna pekare sätts att peka på något annat, så har man tappat det allokerade minnet. Det finns därefter *ingen* möjlighet att hitta det på nytt. Man har *läckt minne*. Se mera info om minnesläckor i Avsnitt 12.3.5.

Vad händer om vårt program allokerar för mycket minne och det tar slut? Detta är ganska sällsynt, men kan hända om minnesläckor förekommer eller om maskinen har litet minne. I så fall kommer `new` att "kasta" en *exception* som heter `bad_alloc`. Mera om exceptions i Kapitel 20, där exempel på en minnes-exception visas.

12.2.2. Allokera vektorer

Förutom att man kan allokera enskilda variabler, såsom en enskild `int`, kan man även allokera en vektor av en och samma datatyp med `new`. Den allmänna formen ser då ut så här:

```
datatyp * pekarnamn = new datatyp [antal];
```

Enda skillnaden från när man allokerar en primitiv är att man specificerar antalet element som vektorn skall innehålla inom `[ ]`. Denna storlek är exakt det antal element i vektorn som kan användas. Exempel på några vektorallokeringar:

```
int * Buffer = new int [1024 * 1024];
float * Koordinat;
Koordinat = new float [3];
string * Brev = new string [100];
```

Man kan allokera mycket stora minnesblock som vektorer. Den enda begränsningen är mängden minne i datorn som används. Det första exemplet allokerar en buffer som är över en miljon element stor, och eftersom en `int` är troligtvis fyra bytes stor (eller större) har man allokerat över fyra miljoner bytes (4 MB). Precis som med normala vektorer bör man inte avreferera element som är utanför det allokerade minnesområdet. Om man t.ex. allokerar 10 element är element 0 det första och element 9 det sista som kan användas, precis som med normala vektorer.

De allokerade vektorerna kan sedan användas precis som normala vektorer, utan speciell avreferering el.dyl., eftersom vektorer i sig redan är pekare. Nedan finns ett exempel som dynamiskt allokerar en vektor och fyller denna med data:

```
#include <iostream>

void in (int * Tal, unsigned int Antal) {
  // iterera och läs in tal
  for ( unsigned int Index = 0; Index < Antal; Index++ ) {
    cout « "Ge in ett tal: ";
    cin » Tal[Index];
  }
}

void ut (int * Tal, unsigned int Antal) {
  // iterera och skriv ut våra tal
```

```
  for ( unsigned int Index = 0; Index < Antal; Index++ ) {
    cout « "Talet är: " « Tal[Index] « endl;
  }
}

int main () {

  unsigned int Antal;
  int * Vektor;

  // hur många tal önskas?
  cout « "Antalet tal i vektorn: ";
  cin » Antal;

  // allokera en vektor för att hålla alla tal
  Vektor = new int [ Antal ];

  // läs in data och skriv sedan ut igen
  in ( Vektor, Antal );
  ut ( Vektor, Antal );
}
```

Processen att hantera dynamiska vektorer är precis likadan som för statiska vektorer, med enda undantaget sättet på vilket de skapas.

12.2.3. Allokering av sammansatta datatyper

Som man kunde gissa är det inget speciellt med att allokera sammansatta datatyper. De fungerar precis på samma sätt som övrig allokering. Om vi t.ex. tittar på den `struct Person` som vi skapade i Avsnitt 9.5. Den såg ut så här:

```
struct Person {
  string Namn;
  string Adress;
  int    Telefonnummer;
};
```

För att allokera ett element av typen `Person` kan man göra följande:

```
Person * P = new Person;
P->Namn = "Joe Foo";
P->Adress = "DataCity";
P->Telefonnummer = 5551234;
```

Noter att vi här använder oss av `->` för att avreferera en pekare till en `struct` istället för avreferering med `*` och punkt. Mera information om detta i Avsnitt 9.5. Man kan även sätta alla värden på samma gång:

```
Person * P = new Person { "Joe Foo", "DataCity", 5551234 };
```

Vi skall nu skapa ett litet program som hanterar en *stack* som innehåller element av datatypen `char`. Vi vill göra stacken totalt dynamisk så att man kan sätta till och ta bort element enligt önskan. Detta exempel visar hur man kan använda pekare tillsammans med `struct`. Varje element i stacken bör innehålla en `char`, samt en pekare till elementet under. Vi definierar detta element på följande sätt:

```
struct Element {
  // data för detta element
  char Data;
  // nästa element under detta i stacken
  Element * Previous;
};
```

Vi har alltid en pekare `Previous` som pekar på elementet under. Vi får således en enkel länkad lista. Första elementet har `Previous = 0`. Då man lägger till ett element kommer det alltid överst på stacken, och dess `Previous` sätts till att peka på det tidigare översta elementet. En enkel stack behöver funktioner för att lägga till ett element på stacken (`push()`), ta bort första elementer (`pop()`) samt en funktion för att testa och stacken är tom (`isEmpty()`). De tre funktionerna kan vi implementera på följande sätt:

```
#include <iostream>

struct Element {
  // data för detta element
  char Data;
  // nästa element under detta i stacken
  Element * Previous;
};

// push:a ett element på stacken
Element * push (Element * Top) {
  char Data;

  cout « "Elementets värde: ";
  cin » Data;
```

```
  // skapa nytt element
  Element * New = new Element;
  New->Data = Data;
  New->Previous = Top;

  // återvänd och returnera nya topp-elementet
  return New;
}

// pop:a ett element från stacken
Element * pop (Element * Top, char & Data) {
  // är stacken tom?
  if ( Top == 0 ) {
    // stacken tom
    return 0;
  }

  // spara data som är lagret i elementet
  Data = Top->Data;

  // spara pekare till det element som blir nytt topp-element
  Element * Tmp = Top->Previous;

  // frigör minne och återvänd
  delete Top;
  return Tmp;
}

// kolla om stacken är tom
bool isEmpty (Element * Top) {

  if ( Top == 0 ) {
   // stacken tom
   return true;
  }

  // stacken inte tom
  return false;
}
```

Koden ovan är relativt enkel att förstå. Både `push()` och `pop()` modifierar stacken, så de returnerar därför den nya stacken. Anroparen måste därför "fånga upp" detta returvärde för att undvika fel. En tom stack har alltid värdet 0, eftersom det då inte finns element allokerade. `pop()` returnerar 0 om stacken blir tom. Funktionerna allokerar minne dynamiskt vid en `push()`och frigör det efter en `pop()` (se Avsnitt 12.2.4 för mera information om frigöring av minne). Vi behöver ännu en funktion `main()` som skall använda vår nya stack. Den kan se ut enligt följande:

```
int main () {
  Element * Top = 0;
  char Choice = ’ ’;
  char Data;

  // iterera tills användaren vill avsluta
  while ( Choice != ’3’ ) {
    cout « "Välj: " « endl « "1 - push" « endl « "2 - pop" « endl;
    cout « "3 - sluta " « endl « "-> ";

    // läs in ett menyval
    cin » Choice;

    // vad vill användaren göra
    switch ( Choice ) {
    case ’1’ : Top = push ( Top ); break;
    case ’2’ :
      // har vi element i listan?
      if ( ! isEmpty ( Top ) ) {
        // ja, poppa och skriv ut det poppade värdet
        Top = pop ( Top, Data );
        cout « "Poppade: " « Data « endl;
      }
      else {
        cout « "Stacken tom!" « endl;
      }
      break;
    }
  }
}
```

Här har vi en `while`-loop som itereras ända tills användaren vill avsluta programmet. En liten meny visas där användaren kan välja att lägga eller ta bort ett element. Då vi tar bort ett element kan vi inte göra det ur en tom stack.

Denna stack är relativt begränsad, eftersom den endast kan hantera data av typen `char`, och koden måste ändras för att kunna användas för andra datatyper. Vi kommer senare att förbättra vårt stack-program och göra det mera flexibelt!

12.2.4. Frigöring av minne

Den största skillnaden mellan normala variabler (även kallade *automatiska*, eftersom de skapas och förstörs automatiskt) är att variabler som allokerats dynamiskt måste även *frigöras*. I detta fall skiljer sig C++ från t.ex. Java, där systemet automatiskt frigör data

som inte används längre. Detta system kallas *garbage collecting*. I C++ är det programmerarens uppgift. Denna uppgift kan vålla en hel del huvudbry för ovana programmerare. Med frigöring av minne menas att ett minnesblock som är allokerat med `new` ges tilbaka till operativsystemet för att användas till något annat ändamål. Om man inte frigör använt minne kommer minnet förr eller senare att ta slut.

I exemplet ovan frigjordes inget minne, är programmet då dåligt? Nej, eftersom operativsystemet automatiskt frigör allt minne som finns allokerat då ett program terminerar. För små program med kort körtid fungerar denna approach. Man kan helt enkelt lämna frigörandet av minne till operativsystemet. Om det däremot är frågan om ett program som kommer att ha en lång körtid (t.ex. en *demon*) eller som använder mycket stora minnesmängder (t.ex. ett grafikhanteringsprogram) är det viktigt att frigöra minne. Mera info om minnesläckor i Avsnitt 12.3.5.

Minne frigörs med operatorn `delete`. Eftersom det är frågan om en *operator* fungerar den på samma sätt som `new`. Allmänt frigörs minne på följande sätt:

```
delete variabel;
delete [] vektor;
```

Man kan alltså frigöra både enkla allokerade variabler och vektorer. Enda skillnaden är att man placerar en tom `[]` efter `delete` då man avser en vektor. Några exempel (vi antar att alla pekare är initialiserade med `new`):

```
string * Text = new string;
delete Text;
int * Buffer = new int [4096];
delete [] Buffer;
```

Notera att en vektor som innehåller endast ett element ändå är en vektor. Operatorn `delete` får endast användas på pekare som har returnerats av `new`, eller så på 0. Om `delete` appliceras på 0 har operationen ingen verkan. Om däremot `delete` appliceras på en pekare med ett slumpmässigt värde uppkommer troligtvis felsituationer. T.ex. följande är fel:

```
int main () {
  int * A;
  delete A;
```

```
}
```

Vi kan inte vara säkra på att `A` har ett definierat värde, eller mera specifikt, att den har värdet 0. Om `A` har värdet 0 så händer ingenting, men om `A` har ett slumpmässigt värde försöker vi troligtvis frigöra minne som inte allokerats. Vi kunde korrigera vårt program från ovan så att det explicit frigör allokerat minne, vilket är bra att bli van med:

```
int main () {
  unsigned int Antal;
  int * Vektor;

  // hur många tal önskas?
  cout « "Antalet tal i vektorn: ";
  cin » Antal;

  // allokera en vektor för att hålla alla tal
  Vektor = new int [ Antal ];

  // läs in data och skriv sedan ut igen
  in ( Vektor, Antal );
  ut ( Vektor, Antal );

  // frigör minne explicit
  delete [] Vektor;
}
```

Endast `main()` visades, de övriga funktionerna är oförändrade.

12.3. Vanliga fel

Det finns ett antal vanliga fel som nybörjare (och experter med för den delen!) inom C++ ofta gör. Några av dessa redogörs för här, för att göra det lättare att veta var felet skall sökas när operativsystemet ger felmeddelandet `segmentation fault` eller `General protection fault`. Problemet med fel inom dynamisk minneshantering är att de kan vara ganska svåra att hitta, eftersom de inte alltid direkt åstadkommer ett

fel, utan ibland kan programmet fungera långa tider trots att ett fel redan uppstått. Sedan kommer en krasch på ett totalt oväntat ställe, kanske på ett ställe som inte ens hanterar dynamiskt minne överhuvudtaget.

12.3.1. Onitialiserat minne

Pekare som skall användas för att peka på allokerat minne kan inte avrefereras före de satts att peka på giltigt allokerat minne. Detta är ett ganska vanligt fel. Följande exempel leder troligtvis till problem:

```
float * Koefficient;

// fel!
*Koefficient = 1.03;
```

Vi har inte allokerat minne för `Koefficient`, inte har vi heller satt den att peka på något annat minnesområde eller någon annan variabel. Kom ihåg att allokera minne före du ämnar använda det!

12.3.2. Frigjort minne

Det är ett allvarligt fel att referera till minne som redan frigjorts. T.ex. följande exempel är felaktigt:

```
double * Buffer = new double [1024];
...
// frigör buffern
delete [] Buffer;

// fel!
Buffer [10] = 34.09;
```

Här försöker vi referera till en vektor vi redan frigjort. I vissa fall kanske detta inte leder till ett omedelbart fel, utan felet kan uppstå senare, speciellt om det minnesblock som frigjordes har återallokerats på annat håll i programmet. I så fall uppstår

minneskorrumption, vilket kan vara *väldigt* svårt att hitta, eftersom felet visar sig på en helt annan plats än det uppstår.

12.3.3. Dubbel frigöring av minne

Allokerat minne får inte frigöras två gånger. Det är frågan om ett allvarligt fel. I ideala fall borde allokerat minne frigöras på endast ett ställe i programmet, men så är inte alltid fallet. Följande program är felaktigt:

```
double * Buffer = new double [1024];
...
// frigör buffern
delete [] Buffer;

....
// fel!
delete [] Buffer;
```

Vi frigör här `Buffer` två gånger. Man kan komma undan dessa problem genom att alltid nollställa pekare efter att minne frigjorts och även då de deklareras. Följande program illustrerar metoden:

```
// initialisera pekare till 0
int * Block = 0;

// allokera minne
...

// kontrollera pekare före frigivning av minne
if ( Block != 0 ) {
  // vi har allokerat minne, frigör och nollställ pekare
  delete [] Block;
  Block = 0;
}
```

Ovanstående metod är mycket bra att använda om man frigör minne på olika ställen i ett program. Man kan då enkelt undvika att frigöra samma minnesblock två gånger.

12.3.4. Fel indexering av vektor

Detta är ett vanligt fel speciellt bland gamla Modula-2 och Pascal-programmerare. I dessa språk kan vektorers övre och undre index sättas separat, medan i C++ är en vektors undre index alltid 0. Man skall således indexera vektorer från 0 till *antalet element - 1*. Följande är en vanlig felsituation:

```
double * Buffer = new double [1024];

// iterera över alla element
for ( int Index = 0; Index <= 1024; Index++ ) {
  Buffer [Index] = ...;
}
```

Vi kommer här att indexera ända till element 1024, vilket är fel, 1023 är det sista elementet som tillhör vektorn. I dylika slingor bör man alltid fundera efter var man skall sluta iterera.

12.3.5. Minnesläckor

Minnesläckor är en annan typ av fel än de som visats ovan. De leder inte till en omedelbar terminering av programmet, utan de äter sakta upp resurser från operativsystemet, och leder förr eller senare till problem då minnet tar slut. I normala fall är små minnesläckor inget större problem, men om det är frågan om program som skall köras länge kan även små läckor leda till stora problem. Antag ett program som läcker minne 1kb (1024 bytes) per minut då det körs. Programmet läcker då 1440kb i dygnet (~1.4Mb). Om programmet skall snurra i dagar och veckor (vilket är normalt under Unix) läcker det 10080kb i veckan och ~43Mb i månaden. På några månader har programmet ätit upp minnet för de flesta maskiner. Antagandet här var att programmet läcker relativt lite minne, om det läckte 1kb i sekunden skulle det läcka ~90Mb i dygnet!

Minnesläckor uppstår då man har allokerat minne och sedan "tappat bort det" utan möjlighet att kunna frigöra det. Följande exempel läcker minne:

```
int * Pekare = new int;
Pekare = new int;
```

Vi har nu inget sätt att veta var vår första `int` finns, eftersom den enda pekare som fanns numera pekar på ett annat minnesområde. Följande är däremot korrekt:

```
int * Pekare1;
int * Pekare2 = new int;
Pekare1 = Pekare2;
int * Pekare2 = new int;
```

Här har vi pekare som pekare på båda minnesområdena vi allokerat, och vi kan således frigöra båda minnesblocken om vi vill:

```
int * Pekare1;
int * Pekare2 = new int;
Pekare1 = Pekare2;
int * Pekare2 = new int;

delete Pekare1;
delete Pekare2;
```

12.4. Minneshantering på C:s vis

Eftersom detta är ett kompendium som behandlar C++ används operatorerna `new` och `delete` genomgående för att allokera och frigöra minne. Dessa operatorer är dock en nyhet som kommit i och med C++. I standard C använder man några funktioner för samma ändamål. Dessa funktioner är dock svårare att använda, och det rekommenderas att man använder C++:s motsvarigheter. Allmänt allokerar man minne enligt följande syntax:

```
datatyp * = (datatyp) malloc (antal_bytes);
```

Funktionen heter `malloc()`, och tar som parameter ett tal som berättar antalet bytes som skall allokeras. För att få reda på antalet bytes en viss datatyp upptar kan man använda `sizeof()`. Ett extra problem är att `malloc()` returnerar en `void *`, d.v.s. en pekare till ett minnesområde utan datatyp. Denna pekare måste typomvandlas (typecast) för att "passa" den pekare som skall användas. För att t.ex. allokera en `int` används föjande:

```
int * Pekare = (int *)malloc ( sizeof (int) );
```

En hel del fulare än:

```
int * Pekare = new int;
```

Det är viktigt att storleken ges rätt, om den är för liten får man nästan garanterat problem då man försöker fylla minnet. Att allokera en `int` enligt nedanstående är helt fel:

```
int * Pekare = (int *)malloc ( 1 );
```

eftersom man då får endast en byte, och en `int` troligtvis behöver fyra. Allokering av vektorer kan man göra genom att multiplicera det antal bytes ett element i vektorn behöver med det antal element man vill ha. Allokering av 100 `char` kan göras på följande sätt:

```
char * Text = (char *) malloc ( sizeof (char) * 100 );
```

Alternativt kan man använda funktionen `calloc`, som tar två parametrar:

```
datatyp * = (datatyp) malloc (antal_element, bytes_per_element);
```

Samma vektor kan då allokeras på följande sätt:

```
char * Text = (char *) calloc ( 100, sizeof (char) );
```

Man bör även under C:s minneshantering frigöra allokerat minne. Detta görs med funktionen `free()`, som allmänt används enligt följande:

```
free (pekare);
```

För att frigöra t.ex. ovanstående `Text` kan man göra på följande sätt:

```
free ( Text );
```

Alla dessa funktioner finns definierade i headerfilen `stdlib.h`, så man behöver därför följande definition i början på varje fil som använder minnesallokering på C:s vis:

```
#include <stdlib.h>
```

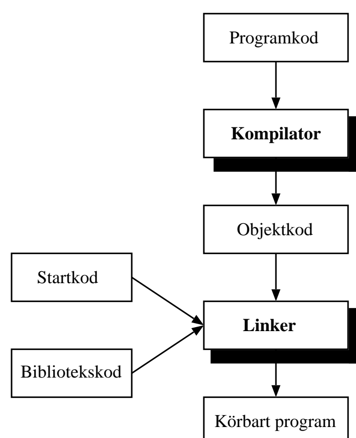
Kapitel 13. Kompilera C++-program

Detta kapitel redogör för hur program kompileras under Unix, eftersom kompilatorer finns fritt tillgängliga till de flesta versioner av Unix. Kompilering under Windows är beroende av om en programmeringsomgivning finns tillgänglig. Alla de olika omgivningarna är olika. Användaren refereras till de medföljande manualerna.

13.1. Kompilatorer

C++-kompilatorer under Unix är reltivt simpla program. Deras enda uppgift är att läsa en programfil där ett C++-program finns och omvandla denna text till ett körbart program. Så gott som alla kompilatorer är *kommandoradskompilatorer*, d.v.s. anändaren ger ett kommando i kommandotolken som startar kompileringen. De uppgifter som kompilatorn gör "bakom kulisserna" kan enkelt visas av följande bild:

Figur 13-1. Kompilatorns arbetsskeden



Den delen som användaren närmast stöter på är den första, själva kompileringsskedet. I

detta skede körs preprocessorn och därefter försöker kompilatorn läsa den fil som skall kompileras och "förstå" den. Men detta avses att kompilatorn parsar koden i filen och bygger upp interna strukturer för hur programmet skall se ut. Om allt verkar vara korrekt och inga syntaxfel finns skapas en fil med *objektkod*. Denna objektkod är programmet assemblerform. Mera info om preprocessorn finns i Kapitel 10. Därefter startas *linkern* som har som uppgift att linka in olika bibliotek, nödvändig extra kod som krävs för att starta programmet sam skriver ut det färdiga körbara programmet. Fel som kan förekomma i detta stadium är t.ex. funktioner som inte finns definierade någonstans eller bibliotek som inte kan hittas på disken. För mera information om bibliotek se Kapitel 11. Vi kommer här att se på hela kompileringsprocessen som en svart låda som antingen kompilerar vårt program eller så ger ett felmeddelande.

13.1.1. Kompilera program

Kompilatorer på Unix har i vissa fall olika namn, men alla fungerar i princip på samma sätt med endast små variationer. Detta kapitel bygger på information om *GNU C++ Compiler*. Denna kompilator heter `g++`. De flesta versioner av Unix har `g++`, men endal har en egen kompilator som ofta heter något i stil med `CC`, `cxx` eller dylikt. `g++` startas genom att skriva `g++` på kommandoraden. Startas den utan argument får man ett kort felmeddelande:

```
% g++
g++: No input files
%
```

Alldeles uppenbart skall man ge en eller flera parametrar till `g++`. Vi antar att hi har ett program i en fil som heter `Hello.cpp`. För att kompilera detta program ger vi kommandot:

```
% g++ Hello.cpp
%
```

Om programmet inte innehåller fel eller underliga konstruktioner fås inga meddelanden överhuvudtaget från kompilatorn. Vi har nu ett program som heter `a.out`, vilket är det

traditionella namnet för namnlösa program under Unix. För att ändra på det namn som kompilatorn ger åt vårt program använder vi switchen `-o` följt av ett filnamn:

```
% g++ Hello.cpp -o Hello
%
```

Det är ingen skillnad i vilken ordning olika switchar ges åt `g++`. Det finns några fall då det är skillnad, men de nämns senare.

13.1.2. Optimering och debuggning

Kompilatorn kan då den genererar den färdiga programmet försöka *optimera* koden, d.v.s. göra den snabbare. Detta betyder inte att kompilatorn försöker skriva om kod som programmeraren skrivit ineffektivt, utan den försöker göra olika slingor så korta som möjligt, undvika långsam kod och använda diverse fula trick för att göra programmet snabbare. Kompilatorn kan i vissa fall försöka om och om igen göra en viss del av programmet snabbare tills den hittar en lösning som är bra. Hur mycket snabbare ett prograam kan bli beror till stor del på typen av program. Om det är ett program som till store del väntar på input från användare eller dylikt kommer en optimering av koden kanppast ens att märkas. Om det däremot är ett program som spenderar en stor del av sin tid i olika beräkningar kan optimering av koden ha en stor inverkan på exekveringstiden.

Kompilering med optimeringar innebär att kompilatorn konsumerar mycket mera minne och tid för kompileringen. Det kan därför löna sig att under utvecklingsskedet kompilera program utan optimeringar, och först då programmet är klart görs en sista kompilering med optimeringar påslagna. Ofta blir de resulterande kärbara filerna en aning större än de skulle blivit utan optimeringar, med skillnaden är oftast mycket liten. För att kompilera med optimeringar använda flaggan `-O` eller `-O`*n*, där *n* är ett värde mellan 0 och 3 (inklusive). Värdet 0 inebär ingen optimering överhuvudtaget och 3 full optimering. För att optimera vårt program `Hello.cpp` kan vi göra följande:

```
% g++ -O2 Hello.cpp -o Hello
%
```

Vi kunde även använda endast `-O`, vilket är ekvivalen med `-O2`.

Om vi har problem med vårt program och vill veta vad det gör finns det två huvudalternativ. Det ena är arbetsdrygt och innebär att man med jämna mellanrum i programmet skriver ut data om vad programmet gör, t.ex. värden på centrala varaiabler. Den andra och mera praktiska metoden innebär att man använder en *debugger*. En debugger kan köra ett program under kontrollerade omständigheter och låter programmeraren titta på variablers värden, köra programmet radvis o.s.v. Det finns olika debuggers under Unix, bl.a. `gdb` (GNU Debugger) och `dbx`. På Linux används `gdb`. För att debuggern skall kunna "debugga" vårt program måste vi instruera kompilatorn att skriva med extra information i det slutgiltiga programmet. Denna extra information innehåller data om variabler, deras typ och namn, själv programkoden som den ser ut i originalfilen o.s.v. Program med debug-information inkluderar är vanligen *mycket* större än program utan samma information. Ofta är det fråga om filstorlekar på tiotals gånger normal storlek. Stora program är långsamma att starta, och ibland kör program med debuginformation även långsammare. Det lönar sig alltså att endast inkludera debug-information då den verkligen behövs. För att göra detta används switchen `-g`. Om vi igen ser på vårt exempel:

```
% g++ Hello.cpp -g -o Hello
%
```

Det är inte heller här någon skillnad vart switchar placeras, så länge de inte placeras mellan en switch och dess argument (t.ex. `-o` och `Hello` ovan).

13.1.3. Varningar

Ofta då man programmerar gör man små fel som egentligen inte direkt är fel, men som kan bli det med lite otur. Man skriver då kod som är tvivelaktig och som borde ändras för att den ska uppfylla standrarderna för C++. I sådana fall kan man få kompilatorn att generera *varningar* för kod som är tvivelaktig. I standardutförande ger t.ex. `g++` mycket få varningar, d.v.s. den tillåter kod som är mycket nära syntaktiskt fel. Om man inte korrigerar dessa olägenheter kanske de orsakar fel senare i programmet. Till `g++` kan man ge switchen `-Wall` för att den skall generera alla möjliga varningar. Det är en

mycket bra sed att alltid se till att alla program kompilerar helt utan varningar, eftersom det är ett tecken på en bra programmerare. Vi kan kompilera vårt `Hello.cpp` med varningar påkopplade, men det ger knappast några fel. Istället kan man prova kompilera nedanstående program, so innehåller två små missar:

```
#include <ostream>

int main () {
  // oanvänd variabel
  int Oanvand;

  enum Alternativ { A, B, C };
  Alternativ Tal = A;

  // switch som ej beaktar värdet 'C'
  switch ( Tal ) {
  case A : cout « "Det är ett A!" « endl; break;
  case B : cout « "Det är ett B!" « endl; break;
  }
}
```

Bara genom att läsa programmet kan det vara svårt att se vad som kunde vara fel på det. Kompilerar men det på normalt sätt får man heller inga varningar. Vi antar att programmet heter `Varning.cpp`.

```
% g++ Varning.cpp -o Varning
%
```

Inga varningar. Lägger vi till`-Wall` får vi följande resultat:

```
% g++ -Wall Varning.cpp -o Varning
Varning.cpp: In function 'int main()':
Varning.cpp:13: warning: enumeration value 'C' not handled in switch
Varning.cpp:5: warning: unused variable 'int Oanvand'
%
```

Nu kanske vi ser vad det är frågan om. Vi har en variabel `Oanvand` som vi nog deklararar, men aldrig använder. Det är orsak till en varning. Vi har även en `switch`-konstruktion som kontrollerar en `enum`. Vi har dock ingen `case` för värdet `C`,

och inte har vi heller en `default` som kunde "fånga upp" alla som inte har en egen `case`. Även detta är värt en varning.

Förutom `-Wall` kan man för `g++` ävan använda switchen `-pedantic`, som gör att kompilatorn är pedantisk då den kontrollerar koden. ger ännu mera potentiella varningar än `-Wall`. Man kan med fördel kombinera dessa två switchar och använda dem samtidigt.

13.1.4. Sökstig för headerfiler

Då man använder sig av externa bibliotek behöver man för det mesta inkludera speciella header-filer. I många fall är dessa header-filer placerade på ett ställe på disken där kompilatorn söker automatiskt, t.ex. `/usr/include`, men alltid är de inte det. I sådana fall måste man kunna berätta åt kompilatorn var den skall söka efter header-filer som inte hittas i normala kataloger. Switchen som gör detta är `-I`*katalog*. Om vi har ett program som heter `StortProgram.cpp` som behöver header-filer som finns i både `/opt/Mesa/include` och `/usr/local/include` kan man ge följande kommandorad:

```
% g++ StortProgram.cpp -I/usr/local/include -I/opt/Mesa/include
%
```

Man kan alltså ge flera `-I` på samma kommandorad. De läggs sedan till till den interna sökvägen för kompilatorn. Notera att inget mellanrum finns mellan `-I` och katalognamnet. Om alla header-filer som `StortProgram.cpp` behöver inte kan hittas ens i dessa kataloger ger kompilatorn ett felmeddelande. Man måste då lägga till ytterligare sökstigar med nya `-I`.

13.2. Länkning av program

Det sista som sker då ett program kompileras är att det *länkas*, d.v.s. all kod skrivs ut till den slutgiltiga filen och diverse extra kod inkluderas. Denna extra kod har som

uppgift att bl.a. starta programmet (se till att funktionen `main()` anropas) och se till att kod finns som kan anropa externa bibliotek (se Kapitel 11 för mera information om externa bibliotek). Om man alltsp använder sig av funktioner som finns i bibliotek som *inte* normalt används av kompilatorn måste man berätta vilka bibliotek som skall länkas in i det slutgiltiga programmet. Om detta inte görs vit kompilatorn inte i vilket bibliotek någon viss funktion finns, och programmet kan således inte köras. Istället ger kompilatorn en felmeddelande. Användaren skall då berätta åt kompilatorn med hjälp av diverse switchar `-l`*bibliotek* vilka bibliotek som skall användas.

Hur man vet vad ett bibliotek heter är lite knepigt. Om man vet att ett program använder funktioner från biblioteken `/usr/lib/libMesaGL.so` och `/lib/libm` skall man ge switcharna `-lMesaGL` och `-lm`. Var kommer då dessa namn ifrån? Jo, man skall från biliotekets filnamn lämna bort hela sökstigen och `lib` från början av filnamnet. Därefter skall man även lämna bort alla versionsnumror och `.so` som finns. Kvar blir bibliotekets *basnamn*. Det enare biblioteket, `libm`, är det normala matematikbiblioteket som innehåller funktioner som t.ex. `sin()`, `log()` och `sqrt()`. Om vi har ett program som heter `Grafik.cpp` som använder båda dessa biblioteken bör vi ge följande kommandorad:

```
% g++ Grafik.cpp -lMesaGL -lm
%
```

I detta fall är det skillnad i vilken ordning switcharna ges. Alla switchar som hör till länkingen av programmet måste komma sist på kommandoraden, efter alla `-O`, `-Wall` o.dyl.

13.2.1. Sökstig för bibliotek

Precis som för header-filer vet kompilatorn för det mesta var olika bibliotek finns på disken då det är dags att länka in dem, speciellt om man använder normala bibliotek som hör till kompilatorn. Om man dock har något speciellt bibliotek man använder som inte finns på ett ställe som kompilatorn söker på måste man berätta åt denna var det finns. Man vet att ett bibliotek inte hittas av att kompilatorn ger ett felmeddelande. Man använder flaggan `-L` för att ge en stig till en katalog där biblioteket finns. Denna flagga fungerar precis som `-I` fungerar för headerfiler (se Avsnitt 13.1.4). Vi antar att vi har

ett bibliotek som heter `libMesaGL` som finns i katalogen `/usr/X11R6/lib`. Då används det på följande sätt:

```
% g++ GfxProgram.cpp -L/usr/X11R6/lib -lMesaGl
%
```

Nu vet kompilatorn var biblioteket finns, samt att det skall länkas in i programmet.

13.2.2. Biblioteksberoenden

För att göra saker och ting ännu mera krångliga då man länkar program är det även skillnad i vilken ordning man ger biblioteken! Detta kan vara väldigt förvirrande för någon som för första gången hanterar olika bibliotek. Tumregeln är att ett bibliotek som kallar på funktioner från ett annat måste komma längre till höger. I detta fall vet man att `libMesaGL` använder sig av matematik, så den kallar därför på funktioner från `libm`, varvid `-lm` skall komma till höger om `-lMesaGL` på kommandoraden. Så länge man har endast ett extern bibliotek är det ingen skillnad, men när man har tio eller mera kan det vara svårt att få dem i korrekt ordningsföljd. Lösningen på detta problem er erfarenhet och några praktiska verktyg som kan användas för att kontrollera biblioteks inbördes beroended (t.ex. `ldd` och `nm`).

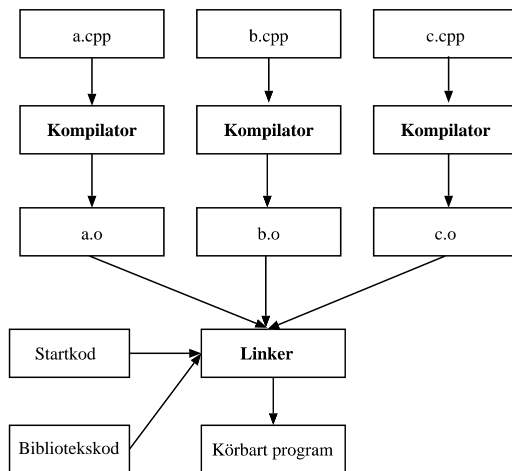
Programmen i detta kompendium kräver dock endast i undantagsfall att man länkar in extra bibliotek, och i så fall är det fårgan om matematikbiblioteket `libm`. Alla program som använder sig av funktioner från `math.h` måste länka in matematikbiblioteket `libm` med hjälp av switchen `-lm`. Placera denna sist på kommandoraden.

13.3. Använda multipla filer

De flesta progam når förr eller senare en kritisk storlek då det inte längre är överskådligt att placera all kod i en och samma fil. Var denna gräns kommer är beroende på programmeraren, men endel personer tycker om att ha många relativt små filer, medan andra föredrar större men färre filer. I C++ finns det förstås möjligheter till detta.

Då man har flera filer som skall i sista hand bli ett och samma program bör de enskilda filerna kompileras skilt för sig till s.k. *objektfiler*, som innehåller koden i assemblerform. När alla filer kompilerats till objektfiler länkas de alla ihop tillsammans med nödvändiga bibliotek och ett körbart program skapas. Exemplet nedan visar schematiskt hur ett program bestående av filerna `a.cpp`, `b.cpp` och `c.cpp` för kompileras till mostavarande objektfiler (`.o`) och sedan linkas till ett körbart program.

Figur 13-2. Kompilatorns arbetsskeden vid multipla filer



För att åstadkomma detta måste man se till att kompilatorn inte direkt försöker länka en fil som kompileras, utan denna skall "lämnas" i objektformatet. Man använder då flaggan `-c`. Man bör då inte ge med flaggor som har att göra med t.ex. bibliotek (`-L` och `-l`) men nog flaggor som har att göra med t.ex. var headerfiler finns. När alla filer kompierats är det dags att länka ihop dem till ett enda exekverbart program. Då ger man alla objektfiler på kommandoraden till kompilatorn och inkluderar eventuella länkningsflaggor, varvid kompilatorn skapar programmet (förutsatt att alllt är korrekt). Ovanstående program kunde kompileras på följande sätt:

```
% g++ -c a.cpp
% g++ -c b.cpp
```

```
% g++ -c c.cpp
% ls
a.cpp a.o b.cpp b.o c.cpp c.o
% g++ a.o b.o c.o -o MittProgram
%
```

13.3.1. Exempel på separat kompilering

Som ett exempel på hur man kan använda separat kompilering och headerfiler i ett eget projekt skall vi titta på ett exempelprogram som definierar en enkelt *stack* (se Avsnitt 12.2.3). Denna stack är simpel och innehåller endast några få funktioner. Vi skall separera implementationen av stacken i en skild fil `Stack.cpp` och definitionen av funktionsprototyper för stacken i `Stack.h`. Sedan skapar vi ett enkelt huvudprogram i en tredje fil `StackMain.cpp` som använder sig av vår stack.

Definitionsfilen `Stack.h` ser t.ex. ut på följande sätt:

```
// Stack.h

// se till att vi inte får problem om filen inkluderas multipla gånger
#ifndef STACK_H
#define STACK_H

// inkludera headers som kan behövas
#include <iostream>

// definiera ett element i stacken
struct Element {
  // data för detta element
  char Data;
  // nästa element under detta i stacken
  Element * Previous;
};

// funktionsprototyper
Element * push (Element * Top);
Element * pop (Element * Top, char & Data);
bool isEmpty (Element * Top);

#endif
```

Filen `Stack.cpp` som innehåller den egentliga koden för alla funktioner ser ut som nedan. Den måste även inkludera `Stack.h` för att `Element` skall vara definierad.

```
// Stack.cpp - implementation av stacken

// inkludera våra egna definitioner
#include "Stack.h"

Element * push (Element * Top) {
  char Data;

  cout « "Elementets värde: ";
  cin » Data;

  // skapa nytt element
  Element * New = new Element;
  New->Data = Data;
  New->Previous = Top;

  // återvänd och returnera nya topp-elementet
  return New;
}

// pop:a ett element från stacken
Element * pop (Element * Top, char & Data) {
  // är stacken tom?
  if ( Top == 0 ) {
    // stacken tom
    return 0;
  }

  // spara data som är lagret i elementet
  Data = Top->Data;

  // spara pekare till det element som blir nytt topp-element
  Element * Tmp = Top->Previous;

  // frigör minne och återvänd
  delete Top;
  return Tmp;
}

// kolla om stacken är tom
bool isEmpty (Element * Top) {

  if ( Top == 0 ) {
   // stacken tom
   return true;
  }

  // stacken inte tom
  return false;
}
```

Själva huvudprogrammet som använder sig av vår stack ser ut som nedan. Den enda skillnaden mellan detta program och det i Avsnitt 12.2.3 är att nu finns hela stacken definierad och implementerad i andra filer, och det enda som behövs är en `#include "Stack.h"`.

```
#include "Stack.h"

int main () {
  Element * Top = 0;
  char Choice = ' ';
  char Data;

  // iterera tills användaren vill avsluta
  while ( Choice != '3' ) {
    cout « "Välj: " « endl « "1 - push" « endl « "2 - pop" « endl;
    cout « "3 - sluta " « endl « "-> ";

    // läs in ett menyval
    cin » Choice;

    // vad vill användaren göra
    switch ( Choice ) {
    case '1' : Top = push ( Top ); break;
    case '2' :
      // har vi element i listan?
      if ( ! isEmpty ( Top ) ) {
        // ja, poppa och skriv ut det poppade värdet
        Top = pop ( Top, Data );
        cout « "Poppade: " « Data « endl;
      }
      else {
        cout « "Stacken tom!" « endl;
      }
      break;
    }
  }
}
```

Programmet kan sedan kompileras med följande kommandon:

```
% g++ -c Stack.cpp
% g++ StackMain.cpp Stack.o
% ./Stack
...
```

Notera att vi här kompilerade och länkade `StackMain.cpp` på samma gång genom att räkna upp de objekt-filer vi ville länka med på kommandoraden. Vi kunde även

kompilerat även denna fil men flaggan `-c` och sedan gjort en separat länkning av programmet.

Kapitel 14. Klasser

Detta kapitel behandlar grunderna i objektorientering i C++. Det vi hittills behandlat av C++ har till största delen varit ämnen som fungerar på samma sätt i standard C (med några undantag).

14.1. Allmänt om objektorientering

De centrala begreppen inom objektorientering är *klass* och *objekt*. Det kan vara värt att redogöra för hur dessa relaterar sig till varandra för att undvika missförstånd i framtiden.

En *klass* definierar en datatyp i C++. Man kan efter att en klass definierats skapa variabler av denna datatyp. Klasser innehåller data och metoder, men en klass i sig kan inte göra något, den endast definierar hur variablerna skall se ut, på samma sätt som `struct`.

Variabler som har en viss klass som datatyp kallas *objekt*. Objekten innehåller egentlig data, och man kan kalla på metoder som objekt har. Man säger att ett objekt är en *instans av en viss klass*. Då man skapar ett objekt *instantierar* man en klass.

14.1.1. Abstraktion

Ett av de centrala målen med objektorientering är *abstraktion*. Man vil dölja komplexitet hos någonting genom att kapsla in fenomenet i en klass. Människan vill alltid ha abstraktioner för olika ting. Vi vill t.ex. inte se på en hand som en samling blod, ben, senor och kött, inte heller som en enorm samling celler av olika typ och allra minst som en ofantlig samling atomer. På samma sätt är vi nöjda över att slippa bry oss om våra program på disken som är en samling data, som är en samling bytes som är en samling bitar som är en samling magnetfält på en skiva. Klasser i C++ skall göra något komplext enkelt att förstå och använda.

Klasser är den mest avancerade abstraktionen i C++. Klasser låter oss abstrahera mycket komplex data på ett intuitivt sätt. Man kan bygga upp logiska arvshierarkier (se Kapitel 15) och kapsla in klasser i varandra. Traditionellt i C har programmeraren alltid vetat vad som finns "bakom" en datatyp, men i C++ behöver inte användaren längre veta detta. Tvärtom kan man totalt abstrahera bort implementationsdetaljer så att programmeraren inte ens kan se dessa, utan endast har ett väl definierat *gränssnitt* att jobba med. Ett gränssnitt definierar hur objekten kan användas.

Förutom endast abstraktion av data kan klasser även tillhandahålla *metoder* som opererar på det data ett objekt innehåller. På detta sätt abstraheras programmeraren från interna implementationsdetaljer angående data ett objekt innehåller, samt för hur denna data skall manipuleras för att åstadkomma ett önskat resultat.

Abstraktionen fungerar även åt andra hållet. Istället för att endast skydda programmeraren från implementationsdetaljer används abstraktion även för att skydda objekt från programmeraren. På detta sätt kan man garantera att ett objekt alltid uppfyller vissa invarianter, eftersom programmeraren inte kan göra något annat med ett objekt än vad dess gränssnitt tillåter. Programmeraren ges inte tillträde till data inom ett objekt om det inte explicit tillåts. Man kan t.ex. tänka sig ett objekt som innehåller interna datastrukturer för ett användargränssnitt. Om programmeraren ges tillträde till alla dessa datastrukturer kan han/hon i misstag göra något fatalt som påverkar även andra programs funktioner negativt.

14.1.2. Återanvändning av kod

Ett av de andra centrala begreppen inom objektorientering är *återanvändning* av kod. C++ gör det lättare att skriva kod som kan återanvändas senare i andra projekt. Tack vare att implementationen är inkapslad behöver man inte veta hur någonting görs, utan det räcker med att veta att det görs korrekt för att klassen skall kunna återanvändas. En bra skapad klass är gjord så att dess funktionalitet kan modifieras via arvning (se Kapitel 15). Det kan kännas svårt för en nybörjare att skapa klasser som är "bra", men det är en teknik som man lär sig med tiden.

14.1.3. Andra fördelar med OO i C++

Objektorientering inom C++ är ett relativt komplext ämne. det finns väldigt mycket olika funktionalitet som i vissa fall kan verka ganska oförklarlig. Inom C++ kan man använda sig av de flesta av de fördelar modern objektorientering för med sig. Man kan t.ex. använda arvshierarkier för att skapa specialiserade klasser med gemensamma gränssnitt (se Kapitel 15). Man kan använda sig av typparametrisering för att åstdakomma generiska klasser (se Kapitel 23) som kan operera på olika datatyper. Överlagring av operatorer (se Kapitel 21) tillåter att klasser konstrueras att likna primitiva datatyper med samma typ av operatorer, eller göra dem lättare att använda. En funktionalitet som kallas exceptions (se Kapitel 20) tillåter avancerad rapportering av fel som gör program lättare att konstruera så att de är robusta. Numera är även Standard Template Library en del av standarden för C++. Den innehåller en stor mängd olika typparametriserade s.k. *containers* och algoritmer som i mycket stor grad gör det lättare att hantera olika typer av data i C++ (se Kapitel 24).

14.1.4. Objektorienterad design

Hur bra och ändamålsenliga klasser skapas är utanför detta kompendiums omfång. Mera information om detta finns i diverse litteratur om objektorientering. Se *Referenser* för referenser till litteratur. De klasser som demonstreras i detta kompendium är relativt enkla och inte alltid helt ändamålsenliga, men de existerar enbart för att visa någon viss aspekt inom C++. När man programmerar i C++ finns det inget som tvingar programmeraren att designa bra klasser, C++ ger endast verktygen för detta. Bra design är något man måste lära sig via försök och misstag. Ju mer man programmerar C++ desto bättre lär man sig visualisera olika klasser för olika entiteter i ett program.

14.2. En första klass

Allmänt definieras en enkel klass i C++ på följande sätt:

```
class namn {
  metoddefinitioner;
  variabeldefinitioner;
};
```

Det viktigaste i en klassdefinition är nyckelordet `class` och `{};`. Mera behövs inte för att definiera en minimal klass som inte innehåller någonting alls. Vi skall nu skapa en första konkret klass i C++. Det är en enkel klass som abstraherar en punkt i en tvådimensionell värld. I många sammanhang behövs en dylik klass och vi kommer att använda den i ett senare exempel. Klassesn `Coordinate` skall innehålla två *medlemmar* som representerar x- och y-koordinaterna.

```
class Coordinate {
public:
  // konstruktor
  Coordinate (float X, float Y) { m_X = X; m_Y = Y; };

  // medlemmar
  float m_X;
  float m_Y;
};
```

Detta är en fungerande lösning men inte en optimal. Vi har här en *konstruktor* som används för att initialisera ett objekt av typen `Coordinate`, samt två medlemmar som innehåller den data vi vill abstrahera, nämligen två värden. Den kod som körs i konstruktorn (mera om det i Avsnitt 14.3) är skriven direkt efter definitionen. Detta är möjligt, och vi får då en Java-liknande syntax, men vi skall senare se hur det är möjligt att flytta ut koden som hör till metoder till skilda platser. För enkla och korta metoder duger denna approach dock fint. Flaggan `public:` betyder att alla definitioner som kommer efter flaggan är öppna för vem som helst att läsa och skriva. Det betyder att våra medlemmar `X` och `Y` kan accesseras av vem som helst. Denna enkla klass kan *instantieras* till objekt i ett program på följande sätt:

```
Coordinate C = Coordinate ( 3, 5 );
```

Dett är hittills det enda sätt på vilket vi kan initialisera en instans av klassen `Coordinate`. Vi skall senare visa hur man kan skapa andra konstruktorer för detta

syfte. För att sedan använda klassen används samma syntax som för `struct`, d.v.s. man accesserar medlemmar och metoder via en punkt (`.`). För att använda vår nya klass kunde vi göra följande:

```
Coordinate C = Coordinate ( 3, 5 );
cout « "Koordinaterna är " « C.m_X « ", " « C.m_Y « endl;
C.X = 3.3;
C.Y = -4.8;
```

De används alltså precis som en `struct`. En klass kan i princip ses på som en utvidgad `struct`. Vi har använt oss av prefixet `m_` för att markera våra medlemmar. Det är inte nödvändigt, men gör det lättare att skilja på medlemmar och parametrar, såsom i konstruktorn. Man kan även använda denna kortare definition för instantiering:

```
Coordinate C ( 3, 5 );
...
```

Där ges parametrarna direkt efter variabelnamnet. Man kan i dessa enkla exempel använda vilken metod som helst, men det finns några argument (som vi tar upp sernare) som förespråkar att man använder den senare metoden.

14.2.1. Dataskydd

Med *dataskydd* avses här möjligheter för klasser att skydda intern data för utomstående. Klassen ovan har inget dataskydd överhuvudtaget. Vi vill att vår klass kan kontrollera hur dess data accesseras. Vi skall nu introducera en ny dataskyddsflagga som kallas *private:* i vår klass:

```
class Coordinate {
public:
  // konstruktor
  Coordinate (float X, float Y) { m_X = X; m_Y = Y; };

private:
  // medlemmar
  float m_X;
  float m_Y;
```

```
};
```

Vi har nu placerat våra medlemmar som *privata* medlemmar. Detta betyder att ingen utomstående kan accessera dem på något sätt. Endast klassen själv kan accessera dem. Hur ska vi nu komma åt dem då? Vi kan introducera några nya metoder för att både läsa och skriva våra värden:

```
class Coordinate {
public:
  // konstruktor
  Coordinate (float X, float Y) { m_X = X; m_Y = Y; };

  // accessera medlemmar
  float x () { return m_X; };
  float y () { return m_Y; };
  void setX (float X) { m_X = X; };
  void setY (float Y) { m_Y = Y; };

private:
  // medlemmar
  float m_X;
  float m_Y;
};
```

Vi har nu ett väldefinierat gränssnitt som används för att accessera våra datamedelemmar. Vi måste nu ändra vårt textprogram ovan till följande:

```
Coordinate C = Coordinate ( 3, 5 );
cout « "Koordinaterna är " « C.x() « ", " « C.y() « endl;
C.setX ( 3.3 );
C.setY ( -4.8 );
```

Notera att accessmetoderna är `public` medan själva data ännu är `private`. På så sätt kan utomstående anropa metoderna, men inte accessera data direkt. Det finns en del nytta med ett dylikt förfarande, bl.a. har vi *datasäkerhet* och *abstraktion*. Användare tvingas accessera objektet via ett väldefinierat gränssnitt som fungerar så att objektet aldrig blir inskonsistent. I detta exempel är det ingen skillnad på vilka värde som ges in, men man kan i dessa metoder t.ex. placera in kontroller för att given data är korrekt.

Man kan i C++ explicit markera att en viss metod inte modifierar ett objekt då den anropas. Detta behövs då man har objekt som är definierade som `const`, och som då inte får anropa metoder som skulle ändra dessa. Vi ser ju enkelt at metoden `x()` inte ändrar objektet för vilket metoden anropas, men vi kan ändå inte anropa metoden för ett `const` objekt. Lösningen blir då att explicit specificera i klassdefinitionen vilka metoder som inte ändrar på ett objekt. I vårt fall är det emtoderna `x()` och `y()`. De övriga ändrar ju någonting. Detta görs genom att sätta en definition `const` efter parameterdefinitionerna till en metod enligt:

```
...
// accessera medlemmar
float x () const { return m_X; };
float y () const { return m_Y; };
...
```

Nu kan dessa metoder bättre användas med `const` objekt.

14.3. Konstruktor och destruktor

Varje klassdefinition har i normala fall en (eller flera) metod som fungerar som *konstruktorer*. En konstruktor är en speciell metod som exekveras då ett objekt skapas, och är menad att ta hand om diverse initialiseringar och parametrar. Normalt används konstruktorer för att ge initiala värden på diverse medlemmar i en klass. Klassen `Coordinate` har en konstruktor definierad på följande sätt:

```
Coordinate (float X, float Y) { m_X = X; m_Y = Y; };
```

Det som skiljer en konstruktordefinition från en vanlig metoddefinition är att den saknar returvärde och har *exakt* samma namn som klassen själv. Då vi skapar ett objekt med t.ex. följande anrop:

```
Coordinate C = Coordinate ( 19.8, 34.11 );
```

Är det konstruktorn som körs. Denna tar två parametrar som sätts som initialvärden för våra medlemmar. Vi kunde även definiera den på följande sätt:

```
Coordinate () { m_X = 0; m_Y = 0; };
```

varvid vi helt enkelt nollställer medlemmarna då klassen instantieras. Vi kan då inte använda samma syntax för att skapa objekt, utan nu måste vi manuellt sätta medlemmarna till önskade värden:

```
Coordinate C = Coordinate;
C.setX ( 19.8 );
C.setY ( 34.11 );
```

Notera att man kan placera en tom parentes efter `Coordinate C = Coordinate;` om man vill visa att konstruktorn inte tar några parametrar överhuvudtaget, men det är inte nödvändigt. Vilket sätt skall man då använda, eller kan man kanske använda båda sätten?

14.3.1. Initialisering av medlemmar

I våra exempel ovan har konstruktorn egentligen intr gjort något mera än kopierat givna parametrar till datamedelemmar. Kopierandet kan göras lättare om man vill. Istället för att i konstruktorns kropp explicit göra ett antal tilldelningar kan de göras direkt vid konstruktordefinitionen. Vi kan skriva om konstruktorerna från ovan till:

```
Coordinate (float X, float Y) : m_X(X), m_Y(Y) { }
Coordinate () : m_X(0), m_Y(0) { }
```

Man placerar alltså direkt efter parentesen ett `:` (kolon) och räknar därefter upp de medlemmar som skall initialiseras och inom parentse de värden de skall få. Den första ger värden från parametrar, medan den andra ger hårdkodade värden. Har man flera initialiseringar placerar man ett kommatecken emellan. Efter initialiseringarna kommer det som normalt skulle komma. I vårt fall behöver vi inte längre någon kodkropp för konsturktorn, eftersom allt arbete redan gjorts.

Initialiseringarna görs *före* den kod som finns i konstruktorn exekveras, så själva kodkroppen kan använda sig av de initialiserade medlemmarna.

14.3.2. Multipla konstruktorer

För att göra klasser lättare och bekvämare atta använda kan man låta dem ha multipla konstruktorer. Meningen är att de skall ta olika parametrar (om alls) så att de kan skapas på olika sätt. Vi kan modifiera vår klass så att den tillåter båda konstruktorerna vi hittills sett:

```
class Coordinate {
public:
  // konstruktorer
  Coordinate (float X, float Y) { m_X = X; m_Y = Y; };
  Coordinate () { m_X = 0; m_Y = 0; };

  // accessera medlemmar
  float x () const { return m_X; };
  float y () const { return m_Y; };
  void setX (float X) { m_X = X; };
  void setY (float Y) { m_Y = Y; };

private:
  // medlemmar
  float m_X;
  float m_Y;
};
```

Nu kan man enkelt skapa objekt av klassen `Coordinate`. Man kan ha hur många konstruktorer som helst, så länge som ett visst set av parametarar endast kan användas av en konstruktor. De måste alltså vara unika. Om det finns två konstruktorer som tar likadana parametrar vet kompilatorn inte vilken som skall väljas och man får ett felmeddelande vid kompileringsskedet.

Om man inte definierar någon konstruktor överhuvudtaget i en klass kommer kompilatorn att skapa en tom konstruktor som inte gör någonting. Detta endast för att

en konstruktor *måste* finnas. En tom konstruktor för `Coordinate` skulle se ut på detta sätt:

```
Coordinate () { };
```

Det är dock bättre att själv definiera denna konstruktor så att man kan vara säker på att medlemmar initialiseras till giltiga initialvärden.

14.3.3. *Copy*-konstruktor

Det finns en annan typ av konstruktor som används förutom de som visats ovan. En speciell typ av konstruktor används vid vissa specialfall då objekt kopieras. Denna konstruktor kallas för (svengelskt sagt) en *copy-konstruktor*. Även detta är en metod som kompilatorn automatiskt genererar om den inte definieras av programmeraren. Denna konstruktor används bl.a. i följande situationer:

- då ett nytt objekt skapas som en kopia av ett annat redan existerande.
- då ett objekt skickas som parameter till en funktion eller metod. Gäller dock ej om det är frågan om en pekare eller referensparameter.

Allmänt ser en copy-konstruktor ut på detta sätt:

```
class namn {
  namn (const namn & variabel);
};
```

Den tar alltså som parameter ett objekt av samma typ som den själv. Objektet är definierat som en konstant parameter (`const`) som inte kan modifieras samt som en referensparameter för att göra anropet effektivare. Vad denna konstruktor skall göra är att den skall kopiera alla värden i *variabel* till "sig själv", alltså generera en kopia av denna. Copy-konstruktorn för vår klass kunde se ut på följande sätt:

```
Coordinate (const Coordinate & Original) { m_X = Origi-
nal.x (); m_Y = Original.y (); };
```

Här kopierar vi helt enkelt värdena som `Original` innehåller. Notera att eftersom `Original` är definierad som ett `const` objekt *kan man inte* anropa metoder för detta objekt som *inte* är definierade som `const`. Detta är orsaken till att vi tidigare definierade `x()` och `y()` som `const`-metoder. Några exempel på situationer då en copy-konstruktor används:

```
Coordinate C1;
// copy-konstruktorer
Coordinate C2 ( C1 );
Coordinate C3 = C1;
Coordinate C3 = Coordinate ( C1 );
```

Medan anropet:

```
Coordinate C1 (10, 10);
```

inte anropar någon copy-konstruktor. Det är alltså frågan om en ganska viktig konsturktor som det i många fall kan vara värt att definiera manuellt och inte lita på den definition som kompilatorn automatiskt ger åt oss. Kompilatorn kommer att kopiera data medlem för medlem från originalet till kopian, vilket ju nog var exakt samma sak som vi gjorde. Vi kunde alltså använt den automatiska definitionen i vårt fall, men i vissa fall kan man få allvarliga fel om man gör detta. Vi kommer att stöta på sådana klasser senare.

Copy-konstruktorn används även då man skickar objekt som värdeparametrar. Ett exempel:

```
#include <iostream>

class Coordinate {
public:
  // konstruktorer
  Coordinate (float X, float Y) { m_X = X; m_Y = Y; };
  Coordinate (const Coordinate & Original) { m_X = Original.x (); m_Y = Original.y (); };
  Coordinate () { m_X = 0; m_Y = 0; };

  // accessera medlemmar
  float x () const { return m_X; };
  float y () const { return m_Y; };
  void setX (float X) { m_X = X; };
  void setY (float Y) { m_Y = Y; };
```

```
private:
  // medlemmar
  float m_X;
  float m_Y;
};

// skriv ut en koordinat
void printCoordinate (Coordinate C) {
  cout « "X = " « C.x () « ", Y= " « C.y () « endl;
}

int main () {
  Coordinate C1 ( 1, 2);
  Coordinate C2 ( C1 );
  Coordinate C3 = C1;

  // skriv ut de olika koordinaterna
  printCoordinate ( C1 );
  printCoordinate ( C2 );
  printCoordinate ( C3 );
}
```

Vi har här en funktion `printCoordinate()` som tar som parameter ett objekt av typen `Coordinate`. Eftersom det skickas som *värdeparameter* kommer copy-konsturktorn att användas då den lokala variabeln `C` i funktionen `printCoordinate()` skapas. Copy-konstruktorn används alltså endast då objekt skickas som värdeparametrar. Om vi istället skirver om `printCoordinate()` till följande kommer vi inte att använda en copy-konstruktor för att skapa `C`:

```
void printCoordinate (Coordinate & C) {
  cout « "X = " « C.x () « ", Y= " « C.y () « endl;
}
```

Vi har nu undvikit en onödig konstruktion av variabeln `C` och även gjort vårt program en aning effektivare. Men i och med att vi skickar parametrarna till `printCoordinate()` som referensparametrar och vi inte har något behov av att ändra på dessa kan vi använda definitionen:

```
void printCoordinate (const Coordinate & C) {
  cout « "X = " « C.x () « ", Y= " « C.y () « endl;
}
```

Vi lägger alltså till definitionen `const` för parameter `Coordinate & C` för att tydligt markera att denna funktion inte modifierar referensparametern (även om den skulle kunna göra det). Det är god sed att använda `const` för parametrar, funktioner och metoder där det är lämpligt för att göra gränssnitt lättare att använda smat tydligare markera ut var objekt kan tänkas bli modifierade.

14.3.4. Destruktor

Vi har hittills sett olika metoder för hur objekt kan skapas och hur konstruktorer används. I C++ finns det även en metod som körs då ett objekt förstörs, nämligen en *destruktor*. Denna exekveras för varje objekt vid en tidpunkt dår objektet redan är "för sent att rädda". Destruktorer skall användas för att göra diverse uppstädning efter ett objekt. Det kan vara frågan om att frigöra allokerat minne, stänga öppnade filer eller frigöra andra resurser. I många fall måste destruktorer användas för att undvika minnesläckor. En destruktor definieras allmänt på detta sätt:

```
class namn {
  ...
  ~namn ();
  ...
};
```

Man har alltså en metod med samma namn som klassen själv, men som har ett ~ framför namnet. En destruktor tar aldrig parametrar och returnerar aldrig något. Då en destruktor är exekverad finns objektet inte längre. Man behöver aldrig manuellt kalla på en destruktor, det görs totalt automatiskt vid rätt ögonblick. En generell regel är att ett objekt förstörs då det går "utanför scope", d.v.s. då det inte längre är aktuellt. I följande exempel förstörs alla de lokala objekten av typ `Coordinate` då funktionen de är definierade i avslutas:

```
void foo (Coordinate C) {
  Coordinate Tmp1, Tmp2;
  ...
}
```

```
int main () {
  Coordinate Point;
  ...
  foo ( Point );
  ...
}
```

De lokala variablerna `Tmp1`, `Tmp2` och `Point` förstörs då funktionerna avslutas, likaså parametern `C` som skapas i `foo()`. `Tmp1` och `Tmp2` skapas på nytt varje gång `foo()` anropas. En destruktor för vår `Coordinate`-klass har egentligen ingenting egentligen att göra, men vi kan skapa en tom destruktor:

```
class Coordinate {
public:
  // konstruktorer
  Coordinate (float X, float Y) { m_X = X; m_Y = Y; };
  Coordinate (const Coordinate & Original) { m_X = Origi-
nal.x (); m_Y = Original.y (); };
  Coordinate () { m_X = 0; m_Y = 0; };

  // destruktor
  ~Coordinate () { };
  ...
};
```

Destruktorn gör ingenting. Vi kan istället se på en enkel vektor vars längd man kan dynamiskt specificera då den skapas. Vektorn har en konstruktor, en destruktor, två metoder för att läsa och skriva ett värde samt en för att returnera storleken.

```
#include <iostream>

class Vector {
public:
  // konstruktor och destruktor
  Vector (unsigned int Size) { m_Data = new int [Size]; m_Size = Size; }
  ~Vector () { }

  // accessmetoder
  void set (unsigned int Index, int Value) { m_Data[Index] = Value; }
  int get (unsigned int Index) const { return m_Data[Index]; }

  // storleken på vektorn
  unsigned int size () const { return m_Size; }
```

```
private:
  // pekare till datablock
  int * m_Data;

  // storlek på vektor
  unsigned int m_Size;
};


int main () {

  // skapa en dynamisk vektor med 100 element
  Vector V ( 100 );

  // fyll den med data
  for ( int Index = 0; Index < 100; Index++ ) {
    V.set ( Index, Index * 10 );
  }
}
```

I konstruktorn allokeras nu ett minnesblock som rymmer `Size` element. Den observante märker att programmet är mycket dåligt skrivet, vi gör inga kontroller överhuvudtaget att givna `Index` är inom rimliga gränser, men det duger för vårt exempel. Tittar man litet närmare på programmet ser man förstås att det allokerar minne med `new` i konstruktorn, men aldrig frigör minnet! Vi har även en tom destruktor som inte gör någonting överhuvudtaget. Hur skall vi kunna undvika att läcka minne om vi skapar och förstör upprepade instanser av `Vector`? Jo, vi kan se till att frigöra minnet i destruktorn. Det är precis dylika uppgifter som skall skötas i en destruktor. Vår nya destruktor blir således:

```
~Vector () { delete [] m_Data; }
```

Här borde även göras kontroller på att `m_Data` verkligen är allokerad, men det lämnas som en övning för läsaren. Nu läcker vår klass inte längre minne.

14.3.5. Mera om copy-konstruktor

Vi har nu fått ett annat problem med vår `Vector`-klass. Vad händer om vi kopierar ett objekt till ett annat, t.ex. på följande sätt:

```
Vector Source ( 10 );
Vector Destination ( Source );
```

Kör vi detta program kommer vi att få en *segmentation fault*, som indikerar att vi gör något fel ved vår minneshantering. Vad går fel och varför? den automatiska copy-konstruktor som kompilatorn skapar oss kopierar alla medlemmar skilt för sig, d.v.s. `m_Size` och `m_Data` kopieras. Dock kopieras endast pekarens värde, inte minnesområdet som pekaren pekar på. Vi får således två pekare som pekar på samma minnesområde, vilket inte i sig är ett problem. Problemet framkommer då objekten förstörs. Eftersom båda pekar på samma område frigör det ena objektet minnet i destruktorn precis som det är menat, men då det andra objektet försöker frigöra samma minne är det reda frigjort och ett fel uppstår. Man kan komma undan detta genom att skapa en copy-konstruktor som ser till att kopiera även det minne som pekaren pekar på, så att det nya objektet pekar på ett eget minnesområde. Vi definieraren copy-konstruktor den enligt följande:

```
class Vector {
  ...
  Vector (const Vector & Original) {
    // allokera minne
    m_Size = Original.size ();
    m_Data = new int [ m_Size ];

    // kopiera minnet
    memcpy ( m_Data, Original.m_Data, sizeof(int) * m_Size );
  }
```

Funktionen `memcpy()` som användes för att utföra kopieringen är en funktion som är inkluderad i och med `#include <iostream>`. Den kopierar data från ett ställe till ett annat, och ersätter i vårt fall en `for`-slinga. Nu har klassen inte längre samma problem med minne som frigörs två gånger. Inte heller läcker det minne. Nu återstår endast felhantering för att göra den komplett.

Den observate märkte säkert att den nya copy-konstruktorn kunde accessera den privata datamedlemmen `m_Data` i `Original`. Hur kan detta gå för sig, den borde väl vara skyddad? Jo, den är skyddad, men endast för objekt av andra klasser, inte för objekt av

samma klass! Detta kan vara bra att veta då man t.ex. skriver copy-konstruktorer eller andra metoder som tar en (eller flera) parameter av samma klass.

14.4. Nästlade klasser

Vi har hittills använt våra objekt i väldigt enkla omgivningar. Vi har inte alls använt oss av *komposition av objekt* (kallas även nästlade klasser). Det innebär att en klass innehåller objekt av samma eller andra klasser som datamedlemmar. Man kan i C++ nästla objekt i princip hur djupt som helst. I större program har man ofta många nivåer av nästlade klasser. Vi skall nu visa hur vi kan nästla in vår `Coordinate`-klass i andra klasser. Som exempel skall vi ta en cirkel som vi definierar som en mittpunkt och en radie. Mittpunkten är ju logiskt att skriva som en koordinat, varvid vi kommer till följande simplifierade cirkel:

```
class Circle {
public:
  // konstruktorer
  Circle (const Coordinate & Origin, float Radius);

  // accessera medlemmar
  const Coordinate & origin () const;
  float radius () const;

  // ändra radie och position
  void setOrigin (const Coordinate & Origin);
  void setRadius (float Radius);

private:
  // radie och position
  Coordinate m_Origin;
  float m_Radius;
}
```

Här har vi nu en medlem `m_Origin` som är av typen `Coordinate`. Det är alltså inget speciellt med att använda klasser i klasser, så länge som alla använda klasserna är

definierade då de används. Notera att ovan har vi ingen *kod* överhuvudtaget för våra metoder. Vi skall ni visa hur den kan definiera separat. Det är det normala sättet att skapa klasser i C++. Java-syntaxen att placera koden direkt efter metoddefinitionen används vanligen endast för små och korta metoder (som dem vi hittills definierat). Klassens egentliga kod definieras nu på följande sätt:

```
Circle::Circle (const Coordinate & Origin, float Radius) {
  m_Origin = Origin;
  m_Radius = Radius;
}

const Coordinate & Circle::origin () const {
  return m_Origin;
}

float Circle::radius () const {
  return m_Radius;
}

void Circle::setOrigin (const Coordinate & Origin) {
  m_Origin = Origin;
}

void Circle::setRadius (float Radius) {
  m_Radius = Radius;
}
```

Vad som kännetecknar detta sätt att skriva metoderna är att man har ett *klassnamn::* framför varje metodnamn för att berätta för kompilatorn till vilken klass denna metod tillhör. Lämnar man bort det prefixet är det frågan om en helt normal funktionsdefinition. På detta sätt kan vi enkelt separera själv klassdefinitionen från implementationen. Klassdefinitionen sätts normalt i en *headerfil* medan implementationen sätts i en `.cpp` -fil. Vi kunde således skapa två filer `Vector.h` som innehåller själva `class`-definitionen och `Vector.cpp` som innehåller implementationen av metoderna. Se Avsnitt 13.3 för mera information om denna modell.

För att bygga ut vårt bibliotek av figurer kunde vi tänka oss geometriska figurer såsom t.ex. fyrkanter, trianglar, linjer och punkter. Vi kommer att återkomma till dessa senare.

14.4.1. Klasser i klasser

Ett annat sätt att se på nästlade klasser är via klasser i klasser. Man kan definiera *lokala klasser* som kan användas inom den klass där den är definierad. Beroende på om klassen definieras i `public:` eller `private:` kan den användas även av externa entiteter.

Ett exempel på interna klasser ges i Avsnitt 17.1.1, men det kräver mera kunskaper än vi har nu, bland annat om dynamisk minneshantering.

Kapitel 15. Ärvning

Detta kapitel behandlar de olika arvsmekanismerna som är tillgängliga i C++. Ärvning kan användas till många olika ändamål i C++.

15.1. Mål med ärvning

Ärvning är en an de grundläggande byggstenarna inom objektorienterad programmering och även inom C++. Att programmera i C++ utan att kunna använda sig av styrkan som ärvning är som att köra en Ferrari på endast första växeln på halv gas. Det är inte detta kompendiums uppgift att försöka förklara teorin och begreppen bakom ärvning. Vi skall se på några av fördelarna med ärvning och några av de olika modellerna för ärvning.

Några av de grundläggande målen med ärvning av klasser är att kunna skapa nya klasser som är antingen *specialiserade* eller *utvidgade* versioner av basklassen. Med specialiserad avses att man begränsar den nya klassens funktionalitet till ett mindre område än den ursprungliga klassens. Om man t.ex. har en klass som definierar en *atom* kan man ärva *subklasser* för olika typer av grundämnen. Man specialiserar då de nya klasserna att fungera endast som ett visst ämne. Det andra stora målet är att skapa utvidgade versioner av basklassen som har ökad funktionalitet för att hanterna andra uppgifter. Man kan t.ex. ärva en *register*-klass från en *lista*. Man ökar då på den funktionalitet som finns i ursprungsklassen och lägger till metoder för att hantera t.ex. olika typer av poster i registret.

15.1.1. Relationer mellan objekt

Det finns olika teoretiska förhållanden mellan klasser i ett program. Det förhållande som ärvning hanterar brukar kalla *is-a* -förhållanden, vilket betyder att den ärvda klassen är av samma typ som ursprungsklassen. Om man t.ex. har en basklass `Frukt` och ur denna ärver de nya klasserna `Banan` och `Appelsin` kan man säga att de nya

klasserna *är* frukter. En `Banan` är även en `Frukt` och har därför samma medlemmar och metoder som denna, som t.ex. vikt och pris.

Andra förhållanden är *has-a* som används för att representera nästlade klasser (containment). T.ex. en klass `Lunch` innehåller kanske en `Banan`, men det är inte vettigt att ärva klassen `Lunch` från `Banan` endast av denna orsak. Däremot är det vettigt att ha en medlem `Banan` i klassen `Lunch`. Förhållandet *has-a* kan dock sällan uppfyllas genom ärvning.

Ett tredje förhållande mellan klasser kallas *is-implemented-as-a*, och betyder att en klass är implementerad genom att använda en viss klass. Man kan t.ex. implementera en stack genom att använda en dynamisk vektor, men en stack är inte en vektor trots att den är implementerad med hjälp av en sådan. I sådana fall kan ärvning användas, men det är inte alltid vettigt. Bättre är även där att använda sig av nästlade klasser och ge klassen `Stack` en privat medlem `Vector`.

Det sista förhållandet är en form av *uses-a*, där en viss klass använder sig av en annan klass för att utföra någonting. Inte heller detta förhållande uttrycks vettigt med hjälp av ärvning. Det enda förhållandet mellan objekt som alltså bör uttryckas via ärvning är *is-a*. Det kan ibland vara svårt att känna igen dessa förhållanden mellan objekt, men det är essentiellt för att undvika konstgjorda ärvningar eller nästlade klasser.

15.1.2. Återanvändning av kod

En av de fundamentala hörnstenarna inom det man vill åstadkomma med OO är *återanvändning av kod*. Varför uppfinna hjulet om och om igen om man kan använda kod som redan finns? Utan objektorientering är återanvändning av kod relativt klumpig och begränsar sig till användning av funktioner ur ett bibliotek och cut’n’paste från existerande program. Genom att planera klasser i ett objektorienterat språk så att de är lätta att återanvända löser många problem och spara mycket tid om det görs rätt. Med återanvändbara klasser kan man bygga upp olika *klassbibliotek* vars klasser andra programerare sedan kan använda eller ärva ifrån.

Det är dock relativt svårt att göra klasser som är enkla att återanvända, men vi skall i detta kapitel gå igenom de olika metoder som finns tillgängliga för ärvning inom C++.

Vi skall även titta på några implementationsdetaljer som man bör vara medveten om då man tänker sig att klasserna skall kunna återanvändas på ett vettigt sätt. Det krävs tyvärr en del övning innan man klarar av att göra bra återanvändbara klasser i C++.

15.1.3. Terminologi

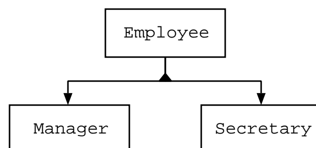
Här finns några begrepp som förekommer runt i texten samlade.

- *basklass* är den klass en viss klass ärver ifrån. Den klass kan ha flera basklasser, ifall det är frågan om en arvshierarki där flera led av ärvning förekommer.
- *subklass* är en ärvd klass. Motsatsen till basklass.
- *superklass* är samma som basklass.

15.2. Konkret exempel

Vi skall nu se på ett enkelt konkret exempel. Vi kan tänka oss att vi vill på något sätt organisera de anställda på ett företag *Foo Factory* i olika klasser med olika uppgifter. Vi tänker oss att den viktiga information vi vill lagra om varje anställd är namn och lön. Förutom normala anställda vill vi även ha med chefer och sekreterare. Vi får då klasshierarkin:

Figur 15-1. Klasshierarki



Vi kan först definiera *basklassen* `Employee`:

```
class Employee {
public:
  // konstruktor
  Employee (string Name, float Salary);

  // returnera data
  string name ();
  float salary ();

private:
  // data om den anställda
  string m_Name;
  float m_Salary;
};

Employee::Employee (string Name, float Salary) {
  m_Name = Name;
  m_Salary = Salary;
}

string Employee::name () {
  return m_Name;
}

float Employee::salary () {
  return m_Salary;
}
```

Då man skapar ene `Employee` ger man med som parametrar till konstruktorn personens namn och lön. Vi har endast en konstruktor och inga möjligheter att ändra vare sig lön eller namn på en anställd. Vi skall nu göra vår första ärvning och skapa classen `Manager` som ärver `Employee`. Vi antar för enkelhetens skull att varje `Manager` är chef för 10 anställda:

```
class Manager : public Employee {
public:
  // konstruktor
  Manager (string Name, float Salary);
```

```
private:
  // de anställda chefen ansvarar för
  Employee m_Employees[10];
};
```

Man definierar alltså ärvning genom att skriva den ärvda klassens namn efter den nya klassens namn. Man sätter ett `:` som separator. Nyckelordet `public` förklaras närmare i Avsnitt 15.4. I övrigt så ser klassdefinitionen helt normal ut. Nu kommer dock vårt första problem då vi ska skriva koden för konstruktorn för `Manager`. Vi måste kunna förmedla parametrar till basklassen `Employee` så vi kan ge värden åt `m_Name` och `m_Salary`. Det finns ett enkelt sätt att skicka med parametrar till en superklass:

```
Manager::Manager (string Name, float Salary) : Em-
ployee (Name, Salary) {
  // initialisera vektor med anställda
  ...
}
```

Man kan alltså skicka med parametrar till en superklass genom att placera ett `:` och klassnamnet efter konstruktorimplementationen. Varför gör man det och när måste man det?

15.2.1. Subklasser och konstruktorer

Då man skapar en subklass skapar man även implicit en basklass. Eftersom en `Manager` är även en `Employee` måste vi se till att vår klass `Manager` även initialiseras att vara en `Employee`. I C++ anropas alltid konstruktorer för basklasser då en subklass skapas. Ordningsföljden är enkel. Först anropas superklassers konstruktorer i ordningsföljd från den första och vidare mot basklassen. I vårt fall anropas konstruktorn för `Employee` först med de parametrar som gavs med, därefeter konstruktorn för `Manager`. Om basklassen inte behöver några parametrar kan man lämna bort `:` och klassnamnet (i vårt fall `: Employee (Name, Salary)`) från implementationen av konstruktorn. Om basklassen däremot kräver parametrar måste man ha en dylik konstruktion. Man måste alltså alltid passa in konstruktorn så att den

passar in med basklassens konstruktor. Basklassens konstruktor körs dock alltid, även om den behöver parametrar eller ej.

Man kan alltså anta och bygga på att konstruktorn för `Employee` har körts då den egentliga koden för konstruktorn för `Manager` körs.

Vi kan nu definiera den tredje klassen arbetare i vårt företag, nämligen `Secretary`. En dylik är även en `Employee`, men har som extra information den `Manager` som han/hon är sekreterare åt. Vi kan säga att klassen `Secretary` *has-a* `Manager`. Vi kan definiera klassen på följande sätt:

```
class Secretary : public Employee {
  // konstruktor
  Secretary (Manager Task, string Name, float Salary);

private:
  // sekreterares 'arbetsuppgift', d.v.s chef
  Manager m_Task;
};

Secretary::Secretary (Manager Task, string Name, float Salary) : Em-
ployee (Name, Salary) {
  m_Task = Task;
}
```

15.2.2. Subklasser och destruktorer

Då objekt förstörs anropas deras destruktorer. I en klasshierarki fungerar det på samma sätt. Jämfört med konstruktorer anropas destruktorer i omvänd ordning, d.v.s. subklassernas destruktorer först, sedan basklassernas. I vårt fall då vi förstör en `Secretary` anropas först destruktorn för `Secretary`, därefter destruktorn för `Employee` och till sist destruktorn för den hypotetiska klassen `Person`. Destruktorer kan alltså även anta att allt som basklassend efinierat finns intakt, medan en basklass vet ett en subklass redan är förstörd då subklassens destruktor anropas.

15.2.3. Anropa subklassers metoder

Vi kan nu skapa en metod som skriver ut lite information om en `Manager`. Informationen skall inkludera namn, lön och namnet på alla `Employee`:s som personen är chef för. Vi behöver då en ny metod till klassen `Manager`, låt oss kalla den `print()`.

```
class Manager : public Employee {
  ...
  // ny metod för att skriva ut information
  void print ();
  ...
}

Manager::print () {
  // skriv ut namn och lön
  cout « "Namn: " « name () « ", lön: " « salary () « endl;

  // skriv ut alla 10 anställda
  cout « "Chef för: " « endl;
  for ( int Index = 0; Index < 10; Index++ ) {
    cout « " m_Employees[Index].name () « endl;
  }
}
```

Vi ser nu att det verkar gå fint för en subklass att anropa metoder i superklassen. Det är exakt så som det är menat, subklasser skall använda sig av den funktionalitet som finns i superklasser. I detta fall är det endast frågan om metoderna `salary()` och `name()` från klassen `Employee`. Eftersom en `Manager` även är en `Employee` kan man för ett sådant objekt även anropa dessa metoder. Om man antar att klassen `Employee` vore ärvd från någon tredje klass, t.ex. `Person`, och där skulle finnas metoden `age()`, kunde även denna metod anropas. Subklasser kan alltid anropa superklassers metoder, om man inte skyddat dem speciellt för att förhindra detta (se Avsnitt 15.4).

15.2.4. Diskussion

Denna klasshierarki är ganska långt värdelös, eftersom klasserna inte innehåller några egentliga metoder för att verkligen göra något, eller en viktig data. Den existerar dock endast för att illustrera hur man kan ärva klasser från varandra.

15.3. Överlagring av metoder

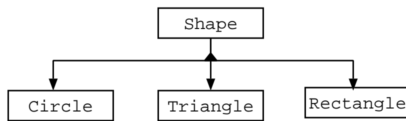
Vi skall nu se på en annan aspekt av ärvning, d.v.s. hur metoder kan överlagras. Med överlagring av metoder avses att en subklass kan ersätta en metod i en basklass med en egen metod, t.ex. för att specialisera metodens beteende till subklassen i fråga. Varför vill man göra något sådant, räcker det inte med att klasser kan anropa metoder ur sina basklasser och skapa nya metoder? Nej, i många fall vill man t.ex. i en basklass skapa ett enhetligt gränssnitt som definierar alla metoder som skall implementeras, och sedan är det upp till de olika subklasserna att implementera dessa metoder på ett sätt som lämpar sig för dem. Om vi tänker oss `Foo Factory`-exemplet ovan kunde vi tänka oss att definiera en metod `void doWork();` för klassen `Employee`. Men eftersom klassen `Employee` inte känner till sna subklasser och därför inte vet hur t.ex. en `Manager` arbetar kan man omöjligt i den metoden ge koden för hur `Manager` skall utföra sitt arbete. Däremot kan metoden innehålla kod för hur `Employee` skall utföra sitt arbete. Därefter kan klasserna `Manager` och `Secretary` *överlagra* metoden `doWork()` och där implementera den kod som representerar det arbete som dessa klasser utför.

15.3.1. Virtuella metoder

För att kunna överlagra metoder behövs i C++ ett koncept som heter *virtuella metoder*. En virtuell metod är en metod som kan överlagras. Inga andra metoder än virtuella metoder kan överlagras. Vi skall senare redogöra för varför det är så. En virtuell metod känneteckans av att den har nycklordet `virtual` före metoddefinitionen. För att titta närmare på virtuella metoder skall vi titta på en ny klasshierarki, nämligen en för att hantera olika geometriska figurer. De figurer vi vill representera är fyrkanter, trianglar och cirklar. Vi kommer att använda oss av klassen `Coordinate` som definierades i Kapitel 14. Eftersom vi vet att varje av dessa figurer är, just det, en figur definieras en

gemensam basklass `Shape` som de olika klasserna ärver. Denna klass innehåller några virtuella metoder som subklasserna förutsätts överlagra.

Figur 15-2. Klasshierarki över figurer



Först definieras klassen `Shape`:

```
class Shape {
public:
  // konstruktor
  Shape (const Coordinate & Origin);

  // virtuella metoder för area och omkrets
  virtual float area () const;
  virtual float circum () const;

  // position för figuren
  const Coordinate & origin () const;
  void setOrigin (const Coordinate & Origin);

private:
  // position
  Coordinate m_Origin;
};

Shape::Shape (const Coordinate & Origin) {
  m_Origin = Origin;
}

float Shape::area () const {
  return 0;
}

float Shape::circum () const {
  return 0;
}

const Coordinate & Shape::origin () const {
  return m_Origin;
}
```

```
void Shape::setOrigin (const Coordinate & Origin) {
  m_Origin = Origin;
}
```

Klassen `Shape` innehåller förutom de två virtuella metoderna `area()` och `circum()` (omkrets) även en konstruktor och en position. Positionen för figuren är en gemensam datamedlem, d.v.s. alla figurer har en viss position, därför är den definierad för klassen `Shape`. Subklasser kan accessera sin position via metoderna `origin()` och `setOrigin()`. Eftersom konstruktorn för `Shape` behöver en parameter måste subklasser anropa denna och ge en `Coordinate` som parameter. Subklasser skall överlagra de virtuella metoderna, därför är de implementerade så att de alltid returnerar 0 för dessa metoder. I framtiden kanske man även behöver olika andra datamedlemmar som är gemensamma för alla subklasser, t.ex. en färg el.dyl., och då kan de placeras i klassen `Shape`. Vi kan om vi vill skapa objekt av typen `Shape`, men de är relativt värdelösa då vi inte kan göra något med dem. Istället definieras en klass `Circle`, som är exakt samma klass som i Kapitel 14, men med de två virtuella metoderna definierade:

```
class Circle : public Shape {
public:
  // konstruktor
  Circle (const Coordinate & Origin, float Radius);

  // överlagrade metoder
  virtual float area () const;
  virtual float circum () const;

  // accessera radien
  float radius () const;
  void setRadius (float Radius);

private:
  // radie
  float m_Radius;
};

Circle::Circle (const Coordinate & Origin, float Radius) : Shape (Origin) {
  m_Radius = Radius;
}

float Circle::radius () const {
  return m_Radius;
}

void Circle::setRadius (float Radius) {
  m_Radius = Radius;
}
```

```
float Circle::area () const {
  // returnera en cirkles area
  return 3.14 * m_Radius * m_Radius;
}

float Circle::circum () const {
  // returnera en cirkles omkrets
  return 2 * 3.14 * m_Radius;
}
```

Metoderna måste i subklasser som ämnar överlagra dem även deklareras och ges typen `virtual`. Man säger då åt kompilatorn att denna klass innehåller en implementation av de virtuella metoderna `area()` och `circum()`. Sedan då de implementeras är det inget speciellt jämfört med implementationen för `Shape`, inga extra definitioner. De skiljer sig dock till den grad att de returnerar en cirkels radie respektive omkrets. Vi definierar ännu en klass för att represententera en rektangel, nämligen `Rectangle`. Vi antar för enkelhetens skull att de rektanglar som används är rätvinkliga.

```
class Rectangle : public Shape {
public:
  // konstruktor
  Rectangle (const Coordinate & Origin,
             const Coordinate & Corner1, const Coordinate & Corner2,
             const Coordinate & Corner3, const Coordinate & Corner4);

  // överlagrade metoder
  virtual float area () const;
  virtual float circum () const;

private:
  // de fyra hörnpunkterna
  Coordinate m_Corners[4];
};
```

Vi implementerar inte dessa metoder, eftersom de är relativt lika de för klassen `Circle`.

15.3.2. Privata metoder

Vi får ett litet problem då vi skall beräkna omkretsen och radien för en `Rectangle`, nämligen hur skall vi få reda på avståndet mellan två punkter så att vi kan beräkna sidornas längd? Vi kan använda Pythagoras för det och göra beräkningen i metoderna

`area()` och `circum()`, men det vore dumt att placera samma kod i två olika metoder. En lösning är att definiera en *privat metod* för klassen `Rectangle` som gör denna beräkning. Vi kan göra följande addition:

```
class Rectangle : public Shape {
  ...
private:
  // beräkna avstånd mellan två punkter
  float distance (const Coordinate & C1, const Coordinate & C2) const;

  // de fyra hörnpunkterna
  Coordinate m_Corners[4];
};
```

Metoden kan sedan implementeras på följande sätt:

```
float Rectangle::distance (const Coordinate & C1, const Coordinate & C2) const {
  float X = C1.x () - C2.x ();
  float Y = C1.y () - C2.y ();
  return sqrt ( (X * X) + (Y * Y));
}
```

Metoder kan alltså definieras som `private` om man vill. En sådan emtod kan endast användas av klassen själv, ingen annan kommer åt den, inte ens subklasser. Privata metoder används vanligen för att flytta ofta använda operationer till skilda metoder, eller för att modularisera något internt förfarande. Vi kommer i framtiden att använda privata metoder nu och då i olika klasser. För att ovanstpende kod skall vara kompilerbar bör man inkludera headefilen `math.h` och länka in biblioteket `libm` som innehåller matematikfunktionerna. Se Kapitel 13 för information om hur man gör detta.

Om man tänker logiskt och objektorienterat hör ovanstående klass egentligen inte hemma i klassen `Rectangle`, utan i klassen `Coordinate`. Den kunde flyttas dit och definieras som en `public` metod. Då bör metoden ändras så att den tar endast en parameter och beräknar avståndet mellan den givna parameterkoordinaten och det aktuella objektet.

15.3.3. Dynamisk bindning

Vi kan för vår figur-klasshierarki tänka oss att skapa en funktion som skulle skriva ut arean och omkretsen för en given figur. Denna skulle vara en fristående funktion som

inte tillhör någon klass. Vi får dok här ett litet probelm. Vi har ju figurer av minst tre olika typer, nämligen `Circle`, `Rectangle` och `Triangle` (ej definierad). Måste vi då ha tre olika funktioner för att skriva ut dessa:

```
void printCircle (const Circle & C);
void printRectangle (const Rectangle & R);
void printTriangle (const Triangle & T);
```

Verkar en aning ineffektivt coh arbetsdrygt. Istället kan vi ta fasta på att alla figurer har basklassen `Shape` och istället använda *dynamisk bindning* av metodanropen för `area()` och `circum()`. Vi definierar funktionen på följande sätt:

```
void print (const Shape & S) {
   cout « "Arean: " « S.area ()
        « ", omkretsen: " « S.circum () « endl;
}
```

Vi tar alltså som parameter en `Shape`. Vi kan då i anropande funktioner t.ex. göra följande:

```
int main () {
  Coordinate Origin ( 10, 10 );
  Coordinate Corner1 (0, 0 ), Corner2 ( 3, 0 );
  Coordinate Corner3 ( 3, 3 ), Corner4 ( 0, 3 );

  // skapa en cirkel och rektangel
  Circle C ( Origin, 5.0 );
  Rectangle R (Origin, Corner1, Corner2, Corner3, Corner4 );

  // skriv ut arean och omkretsen
  print ( C );
  print ( R );
}
```

Här ger vi parametrar av typen `Circle` och `Rectangle` till en funktion som skall ha parametrar av typen `Shape`. Verkar konstigt, eller hur. Kom då ihåg att varje subklass är även av sin basklass' typ, d.v.s. varje `Circle` är även en `Shape`. Tvärtom gäller förstås inte. Därför fungerar detta. När sedan metoderna `area()` och `circum()` exekveras i

`print()` händer något som kallas *dynamisk bindning*. Programmet utför här en kontroll av vilken typ av klass parametern `S` egentligen är, och utför sedan denna metod. Tekniskt sätt så innehåller varje objekt en gömd tabell med adresser till de virtuella funktioner just det objektet skall använda. När en virtuell metod sedan utförs görs en kontroll mot denna tabell och adressen för den korrekta metoden hämtas och metoden utförs. Om en subklass inte har en överlagrad metod innehåller tabellen adressen för basklassens metod. I normala fall då det är frågan om en metod som inte är virtuell görs ingen dylik operation, istället utförs den inkompilerade koden direkt. Det är alltså till viss grad långsammare att ha virtuella metoder p.g.a. denna tabellsökning, men flexibiliteten hos virtuella metoder är enorm. Därför tvingas man i C++ även klart definiera vilka metoder som är virtuella så att kompilatorn vet vilka klasser som skall ha en metodtabell, och vilke metoder som skall palceras i tabellen.

Om en klass inte överlagrar en viss virtuell metod används den senast definierade versionen. Det är alltså inget tvång att överlagra en virtuell metod och den ursprungliga duger.

15.3.3.1. Dynamisk bindning och destruktorer

En klass som har en destruktor där man refererar till den genom dess basklass, måste ha destruktorn virtuell. Vi kan tänka oss ett exempel där vi har en funktion för att radera objekt som allokerats dynamiskt (se Kapitel 17). Den kan se ut på följande sätt:

```
void destroy (Shape * S) {
  delete S;
}
```

Vi kan anta att alla figur-klasses har en destruktor. I koden ovan kommer endast destruktorn för klassen `Shape` att utföras, inte destruktorn för den egentliga subklass som objektet representerar. Om vi skickar en `Circle` vill vi att destruktorn för `Circle` skall utföras först, och därefter destrutorn för `Shape`. Så händer dock inte här. Lösningen är att göra alla *subklassers* destruktorer virtuella. Då utförs de alla och i korrekt ordning. Att göra en destruktorn virtuell innebär endast att nyckelordet `virtual` placeras före destruktordefinitionen.

15.3.4. Abstrakta klasser

Vi har nu tittat på basklasser och olika subklasser och märkt att basklasser är en bra metod för att definiera ett *gränssnitt* som subklasser skall följa, och möjligen ändra funktionalitet för någon metod. Låt oss hypotetiskt tänka att det i en basklass, t.ex. `Shape` finns en eller flera metoder som det helt enkelt inte är möjligt att implementera, för att de är meningslösa på den nivån, eller om vi skulle vilja hindra programmeraren att instantiera objekt av basklassen. I vår enkla figur-hierarki är det fullt möjligt att skapa objekt av typen `Shape`, även om man inte har någon glädje av dem. Det finns ett sätt att hindra detta, samt att inte behöva implementera onödiga metoder. Man kan deklarera en metod att vara `rent virtuell`. En sådan metod innehåller ingen implementation överhuvudtaget. Om vi vill göra metoderna `area()` och `circum()` rent virtuella i `Shape` kan vi göra på följande sätt:

```
class Shape {
public:
  // konstruktor
  Shape (const Coordinate & Origin);

  // rent virtuella metoder för area och omkrets
  virtual float area () const = 0;
  virtual float circum () const = 0;

  // position för figuren
  const Coordinate & origin () const;
  void setOrigin (const Coordinate & Origin);

private:
  // position
  Coordinate m_Origin;
};
```

Enda skillnaden är att vi placerade `= 0` efter metoddefinitionen i klassdefinitionen. Metoderna är nu rent virtuella, och har därför ingen implementationskod i klassen `Shape`.

Det är inte tillåtet att instantiera en klass med rent virtuella metoder. Dylika klasser kallas *abstrakta basklasser*, eftersom de endast kan fungera som bas för andra klasser

som ärer dem och överlagrar de metoder som är rent virtuella. På detta sätt kan vi hindra att programmerare instantierar vår klass `Shape`. Subklasser till en ABK måste inte överlagra alla rent virtuella metoder, men så länge som ens en metod är oimplementerad kan man inte instantiera en klass.

15.4. Typer av ärvning

Den typ av ärvning vi hittills använt är den normala `public` ärvningen. Typen av ärvning specificeras som tidigare nämnts då klassen definieras, t.ex.:

```
class Foo : public Bar {
  ...
};
```

Det finns dock andra typer av ärvning än enbart `public`. De två andra är `private` och `protected`. Typen av ärvning påverkar hur metoder och datamedlemmar är tillgängliga för subklassen samt externa entiteter com använder objekt av klassen.

15.4.1. `public` ärvning

Detta är den normala formen av ärvning. Alla metoder som är definierade som `public` i basklassen är `public` även för subklassen. Data som är deklarerat som `private` i basklassen kan inte accesseras av subklasser eller externa entiteter. I princip ändras inget från hur det är definierat i baskalssen.

15.4.2. `private` ärvning

Denna typ av ärvning är den mest restriktiva formen av ärvning. Den gömmer alla detaljer av basklassen för externa entiteter. Om vi t.ex. ärver klassen `Circle` privat av klassen `Shape` enligt:

```
class Circle : private Shape {
  ...
};
```

då kan ingen som använder objekt av klassen `Circle` referera till metoden `origin()`, eftersom det nu är en privat metod i `Circle`. Privat ärvning kan användas som en form av *has-a* förhållande. Eftersom ingen kan referera till basklassen är det i princip samma sak som om klassen hade haft en dylik datamedlem. Denna typ av ärvning är den minst använda.

15.4.3. `protected` ärvning

Detta är en relativt speciell typ av ärvning. Den är designad för att göra ärvning lätt genom att tillåta basklasser att accessera vissa metoder och medlemmar, medan externa entiteter inte tillåts det. Om vi t.ex. ser på vår `Shape` klass igen märker vi säkert förr eller senare att medlemmen `m_Origin` som innehåller positionen för figuren används ofta av subklasser. Dessa måste varje gång anropa metoderna `origin()` och `setOrigin()` för att accessera denna information. Det vore praktiskt om subklasser istället direkt kunde manipulera `m_Origin` på något sätt. Ett elternativ är att göra den en `public` medlem och sedan ärva `Shape` `public`, men då kan vem som helst manipulera `m_Origin`, vilket inte är bra. Om vi istället gör medlemmen `protected` kan vi använda oss av ett alternativt och bättre sätt. Vi kan då skriva `Shape` på detta sätt:

```
class Shape {
public:
  // konstruktor
  Shape (const Coordinate & Origin);

  // virtuella metoder för area och omkrets
  virtual float area () const;
  virtual float circum () const;

  // position för figuren
  const Coordinate & origin () const;
  void setOrigin (const Coordinate & Origin);
```

```
protected:
  // position
  Coordinate m_Origin;
};
```

Enda skillnaden är flaggan `protected`. I praktiken kan subklasser nu accessera `m_Origin` direkt utan att använda de två accessmetoderna, medan externa klasser och funktioner ännu är begränsade till att använda accessmetoderna. Alla subklasser av subklasser o.s.v. kan accessera `m_Origin` så länge som ärvningen hela tiden är `public` eller `protected`. Man kan alltså även *ärva* protected förutom att ha dataskyddsnivån `protected`. Det kan verka en aning klurigt, och det är det tyvärr också.

Om man ärver en klass som `protected` blir alla metoder och all data som varit `public` till `protected`, medan all data som varit `protected` i baskalssen förblir det i subklassen. Man använder skyddat arv för att införa begränsingar på vad utomstående kan accessera, medan subklasser ännu har full tillgång till allt som varit skyddat som `public` eller `protected` i basklassen. För en utomstående är alltså `protected` arv detsamma som `private`. Det krävs en aning tankearbete innan man fullt förstår hur `protected` arv och dataskydd fungerar.

Man kan göra en basklass *abstrakt* (såsom abstrakta baskalsser med virtuella metoder) genom att göra dess konstruktor `protected`. På så vis kan ingen utomstående accessera konstruktorn, och således ej heller instantiera klassen. Subklasser däremot har full tillgång till konstruktorn och kan anropa den. På så vi kan man instantiera subklasser.

15.4.4. Sammanfattning

Följande tabell sammanfattar hur de olika skyddsnivåerna påverkar de olika arvstyperna.

Tabell 15-1. Skyddsnivåer och arvstyper

	public ärvning	protected ärvning	private ärvning
public medlemmar blir	public medlemmar i ärvda klassen	protected medlemmar i ärvda klassen	private medlemmar i ärvda klassen
protected medlemmar blir	protected medlemmar i ärvda klassen	protected medlemmar i ärvda klassen	private medlemmar i ärvda klassen
private medlemmar blir	ej tillgängliga i ärvda klassen	ej tillgängliga i ärvda klassen	ej tillgängliga i ärvda klassen

15.5. Referera till basklasser

Ibland kan en ärvd klass behöva referera till någon av sina basklasser på något sätt. Är det frågan om metoder som inte är överlagrade som den ärvda klassen kan anropa är det bara att anropa dessa. Vill en klass dock referera till en överlagrad emtod från en basklass är det dock svårare. Om t.ex. `Circle` av någon orsak vill referera till metoden `area()` hos basklassen `Shape` fungerar det inte med att direkt anropa metoden `area()`, eftersom man då anropar den överlagrade versionen av `area()` i `Circle`. Man måste explicit kunna referera till basklassens metod. Det kan man göra genom att placera basklassens namn följt av `::` före metodnamnet. Om vi alltså vill göra ovannämda anrop gör vi följande:

```
void Circle::foo () {
  // anropa area() från Shape
  float Area = Shape::area();
}
```

Notera att denna syntax kan endast användas av objektet själv, d.v.s. det måste göras inifrån en metod som tillhör klassen. Varför vill man göra detta då? Man kanske har en situation där en virtuell metod i en basklass gör någon viss uppgift, och en subklas vill lägga någon viss funktionalitet till. Man överlagrar då metoden i subklassen och implementerar den nya funktionaliteten i denna metod. Basklassens motsvarande metod används ju då inte längre för subklassen. Istället för att kopiera den kod som

basklassen utför till subklassen kan man explicit anropa den överlagrade metoden från basklassen met syntaxen ovan. Beroende på sitautionen kan man anropa metoden t.ex. före den egna extra funktionalitetn utförs. På så vis kan man använda sig av existerande kod på ett snyggt sätt, utan cut'n'paste.

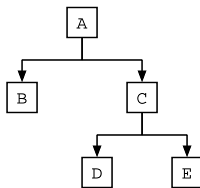
Kapitel 16. Multipel ärvning

Detta kapitel behandlar ett koncept som heter *multipel ärvning*, vilket innebär att klasser ärver från multipla basklasser.

16.1. Vad är multipel ärvning

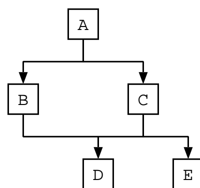
Med *multipel ärvning* avses möjligheten att ärva från flera olika basklasser. Normalt ärver man från endast en basklass och bygger upp en arvshierarki enligt nedanstående figur:

Figur 16-1. Normal arvshierarki



Här ärver alla klasser normalt singulärt. I den modifierade bilden nedan ärver `D` *både* `B` och `C`. `D` är alltså både en instans av klassen `B` och `C`:

Figur 16-2. Multipel arvshierarki



Multipel ärvning används då en ny klass skall representera två eller flera olika andra klasser samtidigt. Vi kan anta att vi har vår gamla klasshierarki från Kapitel 15, där vi introducerade ett företag *Foo Factory* med en arvshierarki av anställda enligt:

```
class Employee { ... };
class Manager : public Employee { ... };
class Secretary : public Employee { ... };
```

Om vi tänker oss att företaget även har en `Technician`-class enligt:

```
class Technician : public Employee { ... };
```

Plötsligt har det uppkommit ett behov för en ny typ av anställd, en person som är chef, men har en teknisk kunskap. Man vill ha en person som är både `Manager` och `Technician`. Lösningen kan i så fall vara att använda multipel ärvning för att skapa denna nya klass:

```
class ManagerTechnician : public Manager, public Technician { ... };
```

För att deklarera att en klass ärver en annan multipelt placerar man alltså flera arvsdeklarationer efter varandra separerade av ett kommatecken (`,`). Man kan ärva från flera klasser än två ifall det finns behov för det. Man kan även ärva klasserna med olika skyddsnivå, även om vårt exempel ärver båda `public.` Nu har vi en ny klass `ManagerTechnician` som innehåller alla metoder från både `Manager` och `Technician`.

16.1.1. Problem med multipel ärvning

Vad finns det då för problem med multipel ärvning? Någonting måste det finnas som gör att metoden används så sällan? Det finns två stora problem som man med stor sannolikhet stöter på, nämligen:

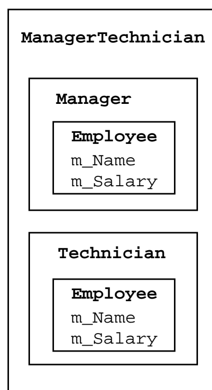
- hur många objekt av basklasserna finns det egentligen? I vårt exempel ärver vi både `Manager` och `Technician`, alltså finns det två olika instanser av `Employee` som ligger i grund för det hela.

- vilken metod körs om samma metod finns i flera olika basklasser? Om våra exempelklasser har metoden `work()` som utför något specifikt för den typen av arbetare, vilken metod skall köras för `ManagerTechnician`? I vårt exempel vill vi sannolikt överlagra metoden `work()`, men om man inte gör det, vad händer då?

16.1.2. Antalet objekt

Vi kan illustrera problemet med flera objekt i den multipelt ärvda klassen med följande figur:

Figur 16-3. Flera objekt i multipel ärvd klass



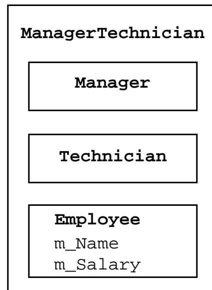
Eftersom både `Manager` och `Technician` ärver `Employee` finns denna klass "dubbelt" i `ManagerTechnician`. Det är nog inte meningen, för en dylik arbetstagare har då t.ex. två namn och två olika löner. Vi skulle vilja att `Employee` skulle kunna vara delad på något sätt. Lösningen på detta problem är att använda sig av *virtuella basklasser*. Vi bör då göra `Employee` till en virtuell basklass för både `Manager` och `Technician`. Nyttan med det är att vid multipel ärvning kommer endast en instans av en virtuell basklass att inkluderas. Vi måste då definiera om våra klasser en aning:

```
class Manager : virtual public Employee { ... };
```

```
class Technician : virtual public Employee { ... };
```

Vi har alltså lagt till nyckelordet `virtual` till klassdefinitionen. Om man ärver multiplet från klasser som alla har en virtuell basklass får man endast en instans av basklasserna, vilket är vad man normalt vill åstadkomma. Vi har nu följande situation:

Figur 16-4. Virtuella basklasser



Klassen `Employee` finns med endast en gång, om personen behöver inte längre lida av personlighetsklyvning. Om man blandar olika typer av arv får man dock ännu samma problem. Om t.ex. `Manager` inte är virtuellt ärvd från `Employee` kommer `ManagerTechnician` att innehålla två instanser av `Employee` igen. Två instanser av samma objekt ger oss probelm om vi t.ex. har en metod som skriver ut löner på arbetare:

```
void paySalary (const Employee & Person) {
  ...
}
```

Vi använder alltså möjligheten att ett objekt alltid kan refereras till via sin basklass. Vilket blir i det senaste fallet det objekt som skickas till `paySalary()`, blir det `Employee`-objektet från `Manager`, eller `Employee`-objektet från `Technician`? Använd därför virtuella basklasser med förstånd där de behövs.

16.1.3. Vilken metod?

Om två klasser som ärvs multipelt av en tredje, och båda har samma metod definierad, vilken metod används då metoden exekveras? Här torde kompilatorn ge ett felmeddelande, eftersom den inte kan vet vilken metod som avses. Man måste då explicit ange vilken metod som avses. Om `Employee` har metoden `work()` så har `ManagerTechnician` samma metod tre gånger definierad, d.v.s. från `Manager`, `Technician` och ursprunsdefinitionen från `Employee` (om det inte var en rent virtuell metod). För att välja den metod som körs bör man använda syntaxen:

```
klassnamn::metod(parametrar);
```

för att explicit ange klassen vars metod skall användas. I vårt fall t.ex.:

```
void ManagerTechnician::doSomething () {
  // utför aktivt ledarskap
  Manager::work ();
  ...
  // utför teknikerarbete
  Technician::work ();
}
```

16.1.4. Diskussion

Vi har sett att det finns en del problem med att använda multipel ärvning i praktiken. Man använder i normala fall mycket sällan multipel ärvning, eftersom de associerade problemen är ganska allvarliga. Purister inom objektorientering anser att multipel ärvning är "evil" och borde helt plockas bort ur C++. I t.e.x. Java finns ingen multipel ärvning såsom i C++, även om man där kan på ett vettigt sätt emulera fördelarna med multipel ärvning via `interface`. För det mesta lönar det sig att tänka om sin klasshierarki om det visar sig att man borde använda multipel ärvning. Exempel i fortsättningen kommer inte att använda sig av multipel ärvning.

Kapitel 17. Mera om klasser

Detta kapitel behandlar olika aspekter på hur klasser fungerar i C++. Många små ämnen har här kombinerats ihop till ett kapitel.

17.1. Dynamiskt allokera objekt

Förutom normala primitiva datatyper, sammansatta datatyper och vektorer kan man även allokera objekt dynamiskt. Dynamiskt allokerade objekt skiljer sig väldigt lite från *automatiska* objekt, d.v.s. objekt som skapas automatiskt av kompilatorn i någon viss funktion eller metod. Fördelarna med att dynamiskt allokera objekt när de behövs är desamma som för andra typer av minne:

- mindre slöseri med minne.
- programmet är mera dynamiskt i olika situationer.

Allmänt ser en allokering av en klass ut på följande sätt:

```
klasstyp * variabel1 = new klasstyp;
klasstyp * variabel1 = new klasstyp (parametrar);
```

Vi antar att vi har en klass dom heter `Coordinate` definierad (se Kapitel 14). Vi kan allokera en dylik genom följande anrop:

```
// använd ursprunsgvärden
Coordinate * C1 = new Coordinate;

// använd egna värden
Coordinate * C2 = new Coordinate ( 1, 2 );
```

Vilken metod som används beror på lklassen ifråga och vilka olika konstruktorer denna har. Vår klass `Coordinate` har både en tom konstruktor och en konstruktor som tar två parametrar. Allokerade objekt måste även förstöras för att inte läcka minne. Det görs

som normalt med operatorn `delete`. För att deallokera ovan allokerade objekt använder vi:

```
delete C1;
delete C2;
```

Man bör minnas att det endas sätt att referera till ett allokerat objekt är via en pekare som pekar till objektet. Finns det ingen pekare längre som pekar på objektet är det förlorat för all framtid och kan inte igen användas. Vi har då läckt minne.

För att allokera en vektor av objekt används samma syntax som för primitiva datatyper:

```
Coordinate * Vector = new Coordinate [42];
```

Man adresserar sedan denna på samma sätt som normala vektorer, d.v.s. via `[]`. En vektor av objekt raderas även på samma sätt som en normal vektor:

```
delete [] Vector;
```

Då objekt allokeras med `new` och förstörs med `delete` exekveras deras konstruktorer och destruktorer på normalt sätt.

17.1.1. Exempel på dynamisk minneshantering

Ett exempel på dynamisk minneshanterin följer i programmet nedan. Det använder sig även av *interna klasser* (se Avsnitt 14.4.1) och illustrerar hur dessa kan användas. Programmet implementerar ett mycket enkelt binärt träd som lagrar heltal. Man kan sätta in tal i trädet samt skriva ut trädet. Trädet använder sig av rekursiva hjälpfunktioner för att skriva ut trädet samt lägga in element.

```
#include <iostream>
#include <stdlib.h>
#include <time.h>

class Tree {
public:
  // konstruktor
  Tree () : m_Root (0) {};
  ~Tree ();
```

```
  // lägg till ett tal
  void add (const int Value);

  // skriv ut trädet
  void print () { print ( m_Root ); }

private:
  // intern klass för att hålla data om en nod
  class Node {
  public:
    Node (const int Value) : m_Value(Value), m_Left(0), m_Right(0) {};
    ~Node ();

    // värdet för noden
    int m_Value;

    // vänster och höger subträdt
    Node * m_Left;
    Node * m_Right;
  };

  // rekursiv metod för att lägga till ett värde och skriva ut
  void add (const int Value, Node * Root);
  void print (Node * Root);

  // roten för trädet
  Node * m_Root;
};

Tree::~Tree () {
  // har vi en rot?
  if ( m_Root )
    delete m_Root;
}

void Tree::add (const int Value) {
  // har vi en rot redan?
  if ( ! m_Root )
    // nej, så skapa en rot och gå iväg
    m_Root = new Node ( Value );

  else
    // lägg till värdet till roten
    add (Value, m_Root);
}

void Tree::add (const int Value, Node * Root) {
  // kolla värdet för denna nod
  if ( Value < Root->m_Value ) {
    // har vi en gren till vänster?
    if ( Root->m_Left ) {
      // jep, rekursera vidare
      add (Value, Root->m_Left);
    }
```

```
    else {
      // nej, så skapa en nod till vänster
      Root->m_Left = new Node (Value);
    }
  }

  // kolla värdet för denna nod
  else if ( Value > Root->m_Value ) {
    // har vi en gren till höger?
    if ( Root->m_Right ) {
      // jep, rekursera vidare
      add (Value, Root->m_Right);
    }
    else {
      // nej, så skapa en nod till vänster
      Root->m_Right = new Node (Value);
    }
  }
}

void Tree::print (Node * Root) {
  // har vi ett värde i noden?
  if ( ! Root )
    // nej, gå bort
    return;

  // skriv ut vänster subträd
  print ( Root->m_Left );
  cout « Root->m_Value « " ";

  // skriv ut höger subträd
  print ( Root->m_Right );
}

Tree::Node::~Node () {
  // har vi ett vänster subträd?
  if ( m_Left )
    delete m_Left;

  // har vi ett höger subträd?
  if ( m_Right )
    delete m_Right;
}

int main () {
  Tree T;

  // initiera slumptalen
  srand ( time (0) );

  // sätt in 20 slumptal i trädet
  for ( int Index = 0; Index < 20; Index++ ) {
    T.add ( (int)((float)rand () / (float)RAND_MAX * 100.0) );
  }
```

```
  // skriv ut trädet
  T.print ();
}
```

Logiken i programmet är relativt enkel och kräver inga närmare beskrivningar. Notera dock hur destruktorn för den interna klassen `Node` implementerats. Eftersom den är intern för klassen `Tree` räcker det inte med enbart:

```
Node::~Node () {
  ...
}
```

utan man måste även berätta att det är en metod för en klass som tillhör `Tree` genom att även skriva `Tree::` enligt:

```
Tree::Node::~Node () {
  ...
}
```

17.2. Friends

Det finns vissa fall då man skulle vilja att klasser skulle kunna bryta mot de skyddsregler som gäller för privata data i klasser. Med detta avses att en viss klass eller funktion skulle kunna accessera de medlemmar och metoder som är deklarerade som `private` i en klass. En situation då ett dylikt förfarande kunde vara praktiskt är då vi t.ex. har en klass `Image` som innehåller en bild, och en klass `ImageViewer` som visar bilder på skärmen. I normala fall kan inga externa exntiter ändra på bildens innehåll, d.v.s. pixeldata, men vi skulle vilja att bildvisaren har vissa möjligheter att korrigera t.ex. storleken eller små fel som uppstår vid förstoring eller rotation av bilden. Det är inte möjligt utan specialarrangemang. Vi kunde tänka oss följande förenklade definitioner på klasserna.

```
class Image {
public:
  // konstruktor
  Image (unsigned int Width, unsigned int Height, int * Pixels);
```

```
  // visa bilden
  void show ();

private:
  // bildens storlek
  unsigned int m_Width;
  unsigned int m_Height;

  // pixeldata (i något internt format)
  int * m_Pixels;
};

class ImageViewer {
public:
  // konstruktor
  ImageViewer ();

  // ladda en bild från disk
  Image * load (string Filename);

  // visa en bild
  void show (const Image & Picture);

  // korrigera en bild enligt någon algorithm
  void fix (Image & Picture);

private:
  // lista av alla bilder
  list m_Images;
};

Image * ImageViewer::load (string Filename) {
  // läs data från filen in i lokala variabler
  unsigned int Width = ...;
  unsigned int Height = ...;
  int * Data = ...;

  // skapa ny bild
  Image * Picture = new Image ( Width, Height, Data );

  // fixa små fel i bilden
  fix ( *Picture );

  // returera den nya bilden vi laddat in
  return New;
}

void ImageViewer::fix (Image & Picture) {
  // accessera pixeldata för 'Picture' enligt fin algoritm
  ...
}
```

Vi går inte närmare in på själva koden för dessa klasser, utan koncentrerar oss på gränssnittet de presenterar utåt. Klassen `Image` har en privat medlem `m_Pixels` som innehåller alla pixlar för en bild. Metoden `fix()` i `ImageViewer` vill kunna direkt accessera pixeldata i bilden `Picture`. Det är inte möjligt, utan kompilatorn kommer att ge oss ett felmeddelande om att vi inte har rätt att accessera privata data i `Image`. Lösningen är att `Image` deklarerar klassen `ImageViewer` som en pålitlig "vän" (friend). En vänklass har rätt att accessera private data i en annan.

Bryter inte detta mot grundprinciperna i objektorientering då? Klassers privata data skall vara privat, anser purister. Inte egentligen eftersom det i detta fall är `Image` som måste explicit säga att `ImageViewer` är dess `friend`. Det går inte andra vägen. Den klass som blir accesserad måste explicit tillåta andra klasser att göra detta. Så det är fråhan om ett medvetet designbeslut man gjort då man skrivit en klass. För att deklarera en klass för en `friend` används förljande allmänna form:

```
friend class klassnamn;
```

som placeras någonstans *inne i* den klass vars privata data skall kunna accesseras av `klassnamn`. I vårt fall kunde vi ändra definitionen av klassen `Image` till:

```
class Image {
  // deklarera en vän
  friend class ImageViewer;

public:
  // konstruktor
  Image (unsigned int Width, unsigned int Height, int * Pixels);

  // visa bilden
  void show ();

private:
  // bildens storlek
  unsigned int m_Width;
  unsigned int m_Height;

  // pixeldata (i något internt format)
  int * m_Pixels;
```

```
};
```

Enda skillnaden är `friend`-deklarationen. Nu kan `ImageViewer` accessera alla privata data i `Image` som om de vore `public`.

17.2.1. Friend-metoder

Om vi tittar lite närmare på vår klass `ImageViewer` märker vi snabbt att det är endast metoden `fix()` som behöver direkt access till `Image`. Vi kunde alltså med fördel begränsa så att endast denna metod kan accessera privata data, istället för att alla metoder kan göra det:

```
class Image {
  // deklarera en vän
  friend class ImageViewer::fix (Image & Picture);

public:
 ...
};
```

Vi skriver nu in hela metodens namn med klassnamn, metodnamn och alla parametrar, exakt som den deklareras i `ImageViewer`. På detta sätt kan t.ex. `load()` inte länge accessera privata data. Det kan vara en god ide att begränsa denna access till exakt de metoder som behöver den, för att inte i misstag i andra metoder accessa privata data när det egentligen inte behövs. Att använda `friend` är inte en lösning på problemet med en dåligt konstruerad klasshierarki. I så fall är det bättre att designa om sina klasser.

En klass kan ha multipla vänner om så skulle behövas. Man placerar bara en ny definition någonstans i klassdefinitionen. Det är ingen skillnad var man placerar `friend`-deklarationer, d.v.s. inom `public`, `protected` eller `private`, så länge som de är separata deklarationer.

17.3. Polymorfism

Det fina ordet polymorfism innebär i praktiken något som kan ses som *överlagring av funktioner och metoder*. Med överlagring avser vi dock inte här i samband med arv, utan på bas av de argument som en funktion accepterar. Vi har redan sett en form av polymorfism i samband med multipla konstruktorer till en klass (se Avsnitt 14.3.2). Multipla konstruktorer ger oss möjligheter att skapa olika typer av konstruktorer för att göra det enkelt att instantiera klassen. Ett exempel på en klass med multipla konstruktorer är:

```
class Line {
public:
  // multipla konstruktorer
  Line ();
  Line (const Coordinate & Source, const Coordinate & Target);
  Line (float X1, float Y1, Float X2, float Y2);

  ...
};
```

Vi kan skapa ett `Line`-objekt på tre olika sätt. Beroende på vilken form av data vi har då objektet skall skapas kan vi troligtvis använda det direkt utan att behöva t.ex. konvertera rena `float`-värden till `Coordinate` eller vice versa. Kompilatorn använder sig av information ur sammanhanget och om de använda parametrarna för att välja vilken konstruktor som används. Om ingen konstruktor finns direkt för de givna parametrarna kollar kompilatorn om det finns något direkt sätt att konvertera de givna parametrarna till någonting som skulle duga. Vi kunde t.ex. instantiera en `Coordinate` med fyra `int`:

```
int X1, X2, Y1, Y2;
Coordinate C ( X1, X2, Y1, Y2 );
```

17.3.1. Funktions- och metodpolymorfism

Förutom endast konstruktorer kan polymorfism användas för normala funktioner och metoder. Detta gör det enkelt att skapa metoder som accepterar argument av olika

datatyp. Om vi tänker oss en funktion som skriver ut data av olika typ på skärmen måste vi utan polymorfism använda metoder med olika namn:

```
void print_int (int Value) { ... }
void print_string (const string & Value) { ... }
void print_Person (const Person & Value) { ... }
```

Det blir ganska fult att göra utskriften på detta sätt. Det är svårt att minnas vilka metoder som kan användas och det är svårare att skriva generiska program. Istället kan man använda sig av polymorfism om skriva funktionerna så här:

```
void print (int Value) { ... }
void print (const string & Value) { ... }
void print (const Person & Value) { ... }
```

En hel del snyggare än det första. Nu behöver man inte längre fundera på vad funktionen heter, eftersom den alltid heter `print()`.

Notera att det inte är möjligt att ha polymorfism med avseende på *returvärdet*. Följande exempel där vi har två metoder, som inte tar några argument, för att skapa objekt av klassen `Circle` och klassen `Rectangel` är inte möjligt:

```
Circle * C = ShapeFactory.create ();
Shape * S = ShapeFactory.create ();
```

Kompilatorn vet inte vilken metod som skall anropas i det senare fallet. Skall man använda metoden som skapar en `Circle` eller den som skapar en `Rectangle`, eftersom båda är möjliga.

17.4. Standardvärden för funktioner och metoder

C++ är väldigt flexibelt med hur parametrar kan ges till metoder. Förutom normal polymorfism (se Avsnitt 17.3) kan man använda sig av standardvärden för parametrar

till funktioner, konstruktorer och metoder. Med standardvärden avses att en eller flera parametrar har ett fördefinierat värde som används ifall anropet inte gett med just den parametern. Ett exempel förklarar saken bättre. Vi kan anta att vi behöver en klass för att lagra en viss färg. Färger brukar vanligen enkodas som ett separat värde på rött, grönt och blått. Man pratar om *RGB-värden*. Mängden av dessa tre komponenter bestämmer den slutliga färgens värde. För att få t.ex. röd färg sätts färgens röda komponent till full intensitet, medan gröna och blå setts till 0. För att få vitt sätts alla komponenter till fullt och tvärtom för svart. Vi kan anta i vårt exempel att intensiteten hos en komponent är mellan 0 och 1.0 (inklusive). 1.0 är full intensitet. Vi kan då skriva en enkel klass för en färg:

```
class Color {
public:
  // konstruktor
  Color (float Red = 0.0, float Blue = 0.0, float Green = 0.0);
  Color (const Color & Original);

  // accessera de enskilda komponenterna
  float red () const { return m_Red; }
  float green () const { return m_Green; }
  float blue () const { return m_Blue; }
  void setRed (float Red) { m_Red = Red; }
  void setGreen (float Green) { m_Green = Green; }
  void setBlue (float Blue) { m_Blue = Blue; }

private:
  // de enskilda komponenterna
  float m_Red;
  float m_Green;
  float m_Blue;
};

Color::Color (float Red, float Blue, float Green) {
  m_Red = Red;
  m_Green= Green;
  m_Blue = Blue;
}

Color::Color (const Color & Original) {
  m_Red = Original.red ();
  m_Green = Original.green ();
  m_Blue = Original.blue ();
}
```

Vi har i den ena konstruktorn använt oss av fördefinierade värden för de olika färgkomponenterna. Vi kan alltså lämna bort dessa parametrar om vi tycker de fördefinierade värdena är bra. Vi kan t.ex. skapa färger på följande sätt:

```
Color White ( 1, 1, 1 );   // full intensitet på alla komponen-
ter blir vit
Color Red ( 1 );           // över-
sätts till: Color Red ( 1, 0.0, 0.0 );
Color Black;               // över-
sätts till: Color Black ( 0.0, 0.0, 0.0 );
```

Den regel som gäller för fördefinierade värde är att man kan lämna bort värde med start från *höger*. Vi kan t.ex. inte tro att objektet `Red(1)` ovan skulle bli t.ex. `Red (0.0, 0.0, 1)`, utan de värden som inte finns lämnas alltid bort från höger. Kompilatorn kan annars inte veta vilket värde man lämnat bort. Man kan i en definition göra endast några värden till fördefinierade:

```
void Image::fill (const Coordinate & UpLeft, const Coordinate & Down-
Right, const Color & FillColor = Color (1, 1, 1));
```

Denna metod kan användas till att fylla en given rektangel i en bild med en viss färg. Här måste man ge parametrarna `UpLeft` och `DownRight`, medan amn kan lämna bort `FillColor` om man är nöjd med den fördefinierade (vit). Notera att även klasser kan vara fördefinierade värden.

Man måste alltid ge åtminstone så många värden till ett anrop som det finns parametrar utan fördefinierade värden. Det är inte heller tillåtet att två funktioner eller metoder "ser lika ut" ifall den ena t.ex. anropas utan värden. Följande är inte tillåtet:

```
class Color {
public:
  // konstruktor
  Color (float Red = 0.0, float Blue = 0.0, float Green = 0.0);
  Color ();
   ...
};
```

Kompilatorn vet inte vilken konstruktor som skall användas om man instantierar ett objekt som `Color NyFarg;`. Båda konstruktorerna kunde användas.

17.5. Statiska metoder och medlemmar

Då vi hittills definierat klasser har vi för det mesta även haft en eller flera datamedlemmar i klasserna. Dessa datamedlemmar finns sedan instantierade i ett objekt. Varje objekt har en egen version av varje medlem. Inget data är delat mellan olika instanser av samma objekt. I vissa fall vore det praktiskt om alla instanser av samma klass kunde dela samma data så att man t.ex. inte behöver flera instanser av en minneskrävande klass. Ett bra exempel då man vill ah en delad variabel är när man vill ge en unik id till alla instanser av en viss klass. Vi kan t.ex. titta på klassen `Product`:

```
#include <iostream>
#include <string>

class Product {
public:
  // konstruktor
  Product (const string & Name, int Id) { m_Name = Name, m_Id = Id; }

  // skriv ut produkten på skärmen
  void print () const;

private:
  string m_Name;
  int    m_Id;
};

void Product::print () const {
  cout « "Produktnamn: " « m_Name « ", id: " « m_Id « endl;
}

int main () {

  Product * Lager [10];
  int Id = 0;
  string Names [10] = {"a", "b", "c", "d", "e", "f", "g", "h", "i", "j"};

  // skapa nya produkter
  for ( int Index = 0; Index < 10; Index++ ) {
    Lager[Index] = new Product ( Names[Index], Id++ );
  }

  // skriv ut vårt lager
  for ( int Index = 0; Index < 10; Index++ ) {
    Lager[Index]->print ();
  }
}
```

Vi använder en lokal variabel `Id` i `main()` för att numrera våra produkter. Egentligen är denna id produktspecifik data, och borde egentligen vara i klassen `Product`. Man hur vi än försöker kan inte klassen `Product` veta vilken som är den senaste använda id:n. Lösningen är att skapa en *statisk variabel* i klassen `Product` som innehåller den senaste id:n. På så vi kan varje instans som skapas läsa denna variabel, kopiera värdet till sin medlem `m_Id` och inkrementera den statiska variabeln. En statisk variabel deklareras med nyckelordet `static`. Vi modifierar `Product` en aning:

```
#include <iostream>
#include <string>

class Product {
public:
  // konstruktor
  Product (const string & Name);

  // skriv ut produkten på skärmen
  void print () const;

private:
  // en statisk variabel
  static int m_NextId;

  // normala medlemmar
  string m_Name;
  int    m_Id;
};

Product::Product (const string & Name) {
  m_Name = Name;
  m_Id = m_NextId++;
}

// initialisera vår statiska variabel
int Product::m_NextId = 0;

void Product::print () const {
  cout « "Produktnamn: " « m_Name « ", id: " « m_Id « endl;
}
```

Vi har nu en privat statisk variabel. den är privat så att inte externa entiteter skall kunna modifiera nästa id, och på så vis eventuellt skapa produkter med samma id. Man måste ge ett initialvärde åt statiska medlemmar, men det kan inte göras inom `class`-definitionen. Man måste då minnas att en `class` endast är en ritning för hur egentliga instanser skall se ut, och inte reservera minne åt medlemmar eller något sådant. Så minne för `m_NextId` kan inte reserveras inne i klassen, det görs en gång

(och endast en gång) då den första instansen av klassen skapas. Initialiseringen sker alltså endast en gång! Vi kan sedan använda variablen inne i klassens metoder som vilken medlem som helst, bara vi minns att om vi ändrar på dess värde syns samma ändring i alla andra instanser av klassen. Vi kan nu modifiera vår `main()` så den använder den nya klassen:

```
int main () {
  Product * Lager [10];
  string Names [10] = {"a", "b", "c", "d", "e", "f", "g", "h", "i", "j"};

  // skapa nya produkter
  for ( int Index = 0; Index < 10; Index++ ) {
    Lager[Index] = new Product ( Names[Index] );
  }

  // skriv ut vårt lager
  for ( int Index = 0; Index < 10; Index++ ) {
    Lager[Index]->print ();
  }
}
```

Nu har vi lyckats förflytta den för `main()` irrelevanta hanteringen av id:n till `Product`.

Statiska medlemmar förstörs då programmet terminerar. Trots att vi skulle radera varje instans av vår `Product`-klass skulle `m_NextId` ändå hålla samma värde.

17.5.1. Statiska metoder

Förutom endast statiska medlemmar i klasser och variabler i funktioner kan man även ha *statiska metoder* i klasser. Vad gör då en statisk metod? En statisk metod är en metod som tillhör klassen med som inte tillhör något instantierat objekt. Den kan inte accessera normala medlemmar som hör till instanser av klassen, utan endast statiska medlemmar. Som exempel kan vi göra en statisk metod till `Product` som skriver ut vad nästa id skall vara:

```
class Product {
public:
  ...

  // statisk metod
  static void printId () { cout « "Nästa id: " « m_NextId « endl; }
```

```
  ...
};
```

En statisk metod definieras genom att placera nyckelordet `static` framför metodens returvärdesdefinition. Metoden kan implementeras antingen som vi gjort ovan direkt inline eller så på normalt sätt:

```
void Product::printId () {
  cout « "Nästa id: " « m_NextId « endl;
}
```

Då man använder metoden kan man inte referera till den via ett objekt, utan direkt via klassen:

```
Product P ("Foo");
P.printId ();        // fungerar inte!
Product::printId (); // korrekt anrop
```

17.5.2. Singleton

Vad har man då för nytta av statiska metoder? I fallet ovan är nyttan klar, men i övrigt finns det inte så väldigt många situationer där statiska metoder är nödvändiga. Ett lysande undantag är en *Singleton*-klass. En dylik klass är en klass som man vill att skall finnas i endast en instans i ett program. Det kan t.ex. vara frågan om en klass som abstraherar en resurs som det finns nedast en av. Istället för att skapa en klass och placera den någonstans så att alla andra klasser som behöver den kan komma åt den, kan man göra den till en Singleton. Det är ett väldigt enkelt och elegant sätt att uppnå en lösning på problemet. Läs mera om Singletons och andra praktiska metoder i *Design Patterns* (se *Referenser*).

17.6. Pekaren `this`

Varje objekt kan referera till sig själv ifall det är nödvändigt. Ibland kanske en objekt vill skicka en pekare till sig själv till någon extern funktion eller metod. Man använder då en speciell pekare som heter `this`. Denna är alltid definierad då man är *inne i* en metod i en klass, i övriga fall är den odefinierad. Man kan alltså inte accessera `this` från t.ex. funktionen `main()`, utan man måste vara inne i en metod. En statisk metod (se Avsnitt 17.5) kan ej heller användas, eftersom man då inte manipulerar ett objekt. Pekaren `this` är alltid en pekare till samma datatyp som klassen man just då "är inne i".

Man kan tänka sig ett osynligt `this` framför alla accesser till medlemmar och anrop av metoder inne i en klass. Om vi t.ex. har vår klass `Coordinate` som normalt ser ut på följande sätt:

```
class Coordinate {
public:
  // konstruktor
  Coordinate (float X, float Y) { m_X = X; m_Y = Y; };

  // accessera medlemmar
  float x () { return m_X; };
  float y () { return m_Y; };
  void setX (float X) { m_X = X; };
  void setY (float Y) { m_Y = Y; };

private:
  // medlemmar
  float m_X;
  float m_Y;
};
```

kan vi tänka oss att metoderna är definierade på följande sätt:

```
  ...
  Coordinate (float X, float Y) { this->m_X = X; this->m_Y = Y; };

  // accessera medlemmar
  float x () { return this->m_X; };
  float y () { return this->m_Y; };
  void setX (float X) { this->m_X = X; };
  void setY (float Y) { this->m_Y = Y; };
  ...
```

Det blir jobbigt att alltid skriva ut `this` på alla ställen så man kan (och gör i de flesta fallen) lämna bort pekaren. Det finns vissa fall då man vill använda sig av pekaren `this`, t.ex. då man vill att en metod skall returnera objektet självt som returvärde.

Kapitel 18. Stränghantering

Detta kapitel går djupare in på hur strängar kan hanteras i C++. Datatypen `string` är mycket flexibel och tillåter många fler operationer än de vi hittills använt.

18.1. Vad är `string` egentligen

Vi har hittills använt `string` som en primitiv datatyp, men det är egenetligen frågan om en ganska avancerad *klass*. Redan av att man alltid måste inkludera headefilen `<string>` för att använda strängar berättar att det är någonting speciellt med `string`.

Vi kan tänka oss att `string` är en vektor av `char`, d.v.s. vanliga tecken. Förutom att den är en abstraktion av en teckenvektor innehåller klassen även diverse metoder och överlagrade operatorer (se Kapitel 21) som gör det enklare att manipulera och använda dem. För dem som använt normala C-strängar av typen `char *` är klassen `string` säkert en välkommen nyhet. Många av de vanligaste felen i gamla C-program härstammar från felaktig användning av C-strängar. Ofta är dessa fel fatala och svåra att lokalisera. Det rekommenderas att man använder `string i` nya program om man inte har tungt vägande skäl till att hålla fast vid C-strängar.

För att kunna använda `string` måste man alltså inkludera headerfilen `<string>`. Utan denna är klassen inte definierad.

18.2. Skapa strängar

Man kan skapa instanser av klassen `string` på flera olika sätt. De vanligaste metoderna är att antingen skapa en tom instans eller så skapa en instans och initiera den direkt med en sträng:

```
string variabelnamn1;
string variabelnamn2 = "värde";
```

Den första deklarationen skapar en tom sträng medan den andra skapar en sträng med ett fördefinierat värde. En kopia av det fördefinierade värdet görs. Strängar kan även skapas som kopior av varandra, enligt följande:

```
string S1 = "Bilbo";
string S2 = S1;
```

Strängen `S2` är nu en *kopia* av `S1`, och all manipulering av `S2` påverkar inte `S1`. Ifall man vill skapa en ny sträng och fylla denna med ett antal av ett visst specifikt tecken kan detta även göras. För att skapa en sträng innehållande 10 *A* kan följande användas:

```
string S ( 10, 'A' );
```

Ibland kan det vara användbart att kunna skapa en sträng av en del av en annan sträng. Man använder då ordet *substräng*. En substräng kan skapas genom att till konstruktorn för `string` ge som parametrar den ursprungliga strängen som man vill ha en substräng ur, indexet där man vill börja substrängen samt antalet tecken. Ett exempel på en substrängsinitiering:

```
string S1 = "Hello world";
string S2 ( S1, 6, 5 );  // starta från 'w' och kopiera fem tecken
```

Även då man initierar från en substräng görs en kopia av tecknen. I exemplet ovan har `S2` ingenting längre gemensamt med `S1` efter initierringen, annat än att den råkar innehålla delvis samma text som i `S1`.

18.2.1. Längden av strängar

För att få reda på längden av en sträng kan man använda sig av metoderna `size()` eller `length()`. Båda ger samma värde, d.v.s. antalet tecken i strängen. T.ex.:

```
#include <iostream>
#include <string>

int main () {
  string S1 = "Hello world";
  string S2;
```

```
  // skriv ut strängens längd
  cout « "size()   : " « S1.size () « endl;
  cout « "length() : " « S1.length () « endl;
}
```

Man kan alltså använda antingen `length()` eller `size()`. En tom sträng har alltid längden 0.

18.3. Accessera enskilda tecken

Man kan enkelt accessera enskilda tecken i en `string`. Ofta vill man t.ex. iterera över tecken och söka efter något visst tecken eller utföra någon annan operation som kräver att man direkt kommer åt de enskilda tecknen. Klassen `string` har två olika metoder för att utföra detta. Det första sättet använder sig av metoden `at()` som returnerar en referens till ett tecken på ett givet index. Ett exempel på att använda `at()`:

```
#include <iostream>
#include <string>

int main () {
  string S1 = "Hello world";

  // ändra 'w' till 'W'
  S1 [6] = 'W';

  cout « S1 « endl;
}
```

Samma exemempel men med `at()` ser ut på följande sätt:

```
#include <iostream>
#include <string>

int main () {
  string S1 = "Hello world";
```

```
  // ändra 'w' till 'W'
  S1.at (6) = 'W';

  cout « S1 « endl;
}
```

De två olika metoderna har en fundamental skillnad. Metoden `at()` kontrollerar om det givna indexet är giltigt, d.v.s. att det ligger inom intervallet 0 till `length() - 1`. Strängar indexeras precis som normala vektorer från 0. Ett försök att accessera ett tecken förbi slutet av strängen leder då man använder `at()` till att en *exception* `out_of_range` kastas. Se Kapitel 20 för mera information om exceptions. Använder man `[]` för att indexera strängen görs ingen dylik kontroll och diverse fel kan uppstå av att accessera förbi slutet på en sträng. Om man dock är noggrann med att inte accessera förbi slutet med `[]` är det ingen skillnad på funktionaliteten på de två sätten.

18.4. Tilldelning och jämförelse

Såsom vi redan sett i tidigare exempel är det möjligt att tilldela strängar värden av andra strängar med hjälp av operatorn `=`. Den tilldelade strängen gör då en *kopia* av originalsträngen. Ett alternativt sätt är att använda metoden `assign()` som kan ta som argument en annan sträng, en C-sträng (`char *`) eller ett tecken (`char`). Den kan användas på t.ex. följande sätt:

```
string S1;
string S2;
S1 = "C++ är kul";

S2 = S1;
// eller
S2.assign ( S1 );
```

Man kan även jämföra strängar med hjälp av operatorn `==`, varvid alla tecken i de två strängarna jämförs och uttrycket evalueras till `true` om de är identiska och `false` och

de ej är det. Man kan alltså skriva kod som i exemplet nedan som jämför en sträng med en strängkonstant:

```
string Anvandare;
cin » Anvandare;

if ( Anvandare == "root" ) {
  // superuser, gör nåt kul
  ...
}
```

Normala vektorer kan man inte jämföra med `==`, men för klassen `string` är det alltså möjligt. Förutom likhet kan man test olikhet m.h.a. operatorn `!=`.

Alfabetisk jämförelse av strängar kan göras med operatorerna `<`, `>`, `<=` och `>=`. Dessa jämför strängarna enligt ASCII-tabellen och evalueras till `true` eller `false` beroende på vilken sträng som är större eller mindre.

18.5. Insättning av text

En vanlig operation då man hanterar strängar är att lägga till text i en sträng. De två möjligheterna här är att text läggs till på slutet av strängen eller någonstans mitt i.

Vi tittar först på hur text kan läggas till på slutet av en sträng. Man kan göra detta med metoden `append()` eller med operatorn `+=`. Båda ändrar strängen och lägger till den givna texten eller tecknet till slutet av strängen. Ett exempel på en funktion som konkatenerar två strängar med ett tomrum emellan:

```
string fulltNamn (const string & Fornamn, const string & Efternamn) {
  string Namn = Fornamn;
  Namn += ' ';
  Namn += Efternamn;

  // ge tillbaka det fullständiga namnet
  return Namn;
}
```

Här används operatorn `+=` för konkateneringen, men `append()` kunde lika väl ha använts. Vi kan konkatenera instanser av `string`, C-strängar och vanliga tecken (`char`).

Att lägga till text på annan plats i en sträng än på slutet görs med metoden `insert()`. Även denna finns i diverse olika varianter. Man kan sätta in var som helst i en sträng instanser av `string`, C-strängar, tecken, substrängar samt ett antal av samma tecken eller C-sträng. Ett exempel:

```
#include <iostream>
#include <string>

int main () {
  string Namn = "Frodo Bagger";

  // lägg till initialen för mellannamnet
  string Initial = "T. ";
  Namn.insert ( 6, Initial );

  // sätt in i början av strängen
  Namn.insert ( 0, "Mr. " );

  cout « "Namnet är: " « Namn « endl;
}
```

Körning av programmet ovan ger:

```
% ./StringInsert
Namnet är: Mr. Frodo T. Bagger
%
```

18.5.1. Addera strängar

Man kan förutom att direkt konkatenera och sätta in text i strängar även addera dem med hjälp av operatorn `+`. De strängar som används i additionen påverkas inte på något sätt. Ett exempel belyser hur detta fungerar:

```
#include <iostream>
```

```
#include <string>

string fulltNamn (const string & Fornamn, const string & Efternamn) {

  // ge tillbaka det fullständiga namnet
  return Fornamn + " " + Efternamn;
}

int main () {
  string Namn = fulltNamn ( "Frodo", "Bagger" );
  string Text = "Fullständigt namn: " + Namn;

  cout « Text « endl;
}
```

Vi kan alltså addera ihop flera olika strängar och stränkonstanter med operatorn + och sedan tilldela strängen till en annan variabel. Denna funktionalitet är praktisk då man t.ex. vill skapa olika meddelanden som skall visas åt användaren.

18.6. Söka text i strängar

En vanlig operation som man kan tänkas utföra på en sträng är att söka om en viss substräng finns i en annan sträng, och i så fall på vilket index den börjar. Det finns ett stort antal olika metoder för sökning av substrängar i strängar och följande avsnitt går igenom dessa.

18.6.1. Sökning från början av strängen

Om man behöver hitta en viss substräng i en sträng kan man använda metoden `find()`. Denna returnerar ett giltigt index (av typen `string::size_type`) där strängen börjar om substrängen kunde hittas eller konstanten `string::npos` om substrängen inte kunde hittas. Detta index kan användas som ett normalt heltal.

Man kan som tidigare söka efter `string`, C-strängar och tecken. Ett index kan ges med som en andra parameter. Detta index berättar på vilket index i strängen sökningen skall startas. Några exempel på sökningar:

```
string Text = "The Lord of the Rings";
string S = "Rings";

string::size_type I1 = Text.find ( "Lord" );  // I1 == 4
string::size_type I2 = Text.find ( "lord" );  // I2 == string::npos
string::size_type I3 = Text.find ( ' ' );     // I3 == 3
string::size_type I4 = Text.find ( S, 10 );   // I4 == 16
```

Alla varianter av `find()` stannar vid den första substräng räknat från början som passar. Senare substrängar som även passar ignoreras.

18.6.2. Sökning från slutet av strängen

Sökning från slutet av en sträng fungerar på samma sätt som sökning av en substräng från början av en sträng (se Avsnitt 18.6.1), med den skillnaden att nu startar sökningen från slutet av strängen. Metoden heter `rfind()` och tar samma argument som vanliga `find()`. Sökningen stannar vid den första hittade substrängen.

18.6.3. Sökning av första och sista förekomst av tecken

Förutom substrängar kan man även söka efter vissa tecken. Detta kan vara praktiskt om man t.ex. behöver dela upp en sträng i delar enligt något tecken. Man kan antingen söka på bas av ett enda tecken eller något tecken i en given sträng. T.ex.:

```
string Text = "The Lord of the Rings";
string S = "ABCL";

string::size_type I1 = Text.find_first_of ( 'e' );   // I1 == 2
string::size_type I2 = Text.find_first_of ( " or" ); // I2 == 3
string::size_type I3 = Text.find_first_of ( S );     // I3 == 4
```

De givna strängarna söks alltså inte i sin helhet, utan de första förekomsten av något av teckne räknas. T.ex. får `I2` värdet 3 trots att texten inte i sin helhet finns i de ursprungliga strängen. Dock finns `' '` (mellanslag) på position 3, så detta index returneras. Den första giltiga positionen returneras.

Man kan även söka efter det *sista* tecknet som finns i strängen som matchar något av de sökta tecknen. Man kan se på detta som att söka från slutet av strängen och returnera den första positionen som passar. Metoden för att göra detta heter `find_last_of()`, och tar samma argument som `find_first_of()`.

18.6.4. Sökning av första och sista förekomst av tecken inte i söksträngen

Det finns två metoder som kan användas för att söka efter tecken i en sträng som *inte* finns i söksträngen. Dessa heter `find_first_not_of()` och `find_last_not_of()`. Dessa söker den första respektive sista positionen i strängen där ett tecken finns som inte finns i söksträngen. Exempel:

```
string Text = "The Lord of the Rings";

string::size_type I1 = Text.find_first_not_of ( "The" );  // I1 == 3
string::size_type I2 = Text.find_first_not_of ( "the" );  // I2 == 0
string::size_type I3 = Text.find_last_not_of ( "Rings" ); // I3 == 15
string::size_type I4 = Text.find_last_not_of ( "abc" );   // I4 == 20
```

18.7. Ersätta text

Man kan enkelt *ersätta* en viss del av en sträng med en annan sträng m.h.a. metoden `replace()`. Denna tar som argument ett index som anger var man vill börja ersätta text, ett tal som anger antalet tecken som skall ersättas och sist den nya texten. Texten kan ges som en `string` eller C-sträng. Exempel:

```
#include <iostream>
#include <string>

int main () {
  string Text = "The Lord of the Rings";

  Text.replace ( Text.find ( "Lord"), 4, "lord" );

  cout « Text « endl;
}
```

Programmet ersätter texten `Lord` med `lord`. Här används en sökmetod för att först hitta den korrekta positionen av ursprungstexten.

18.7.1. Radera text

Ett specialfall av att ersätta text är då man ersätter text med en tom sträng. Vad man i praktiken gör då är ju att man *raderar* text ur strängen. Man kan göra en radering genom att göra en ersättning med `replace()` och specificera en tom sträng (`""`) som den ersättande texten. Alternativet är att använda `erase()`. Denna metod tar två eller inga argument:

```
#include <iostream>
#include <string>

int main () {
  string Text = "The Lord of the Rings";

  Text.erase ( 0, 4 );

  cout « Text « endl;
}
```

Vi raderar alltså fyra tecken med början från det första. Ger man inga argument alls till `erase()` töms hela strängen.

18.8. Substrängar

Vi har hittills använt substrängar för att söka och ersätta text i strängar. Vi kan även returnera godtyckliga substrängar i en sträng. Detta görs m.h.a. metoden `substr()`. Denna tar två parametrar som indikerar positionen där substrängen skall börja respektive längden på substrängen. Om den andra parametern lämnas bort returneras resten av strängen. T.ex.:

```
#include <iostream>
#include <string>

int main () {
  string Text = "The Lord of the Rings";

  string Sub2 = Text.substr ( 9, 2 );
  string Sub1 = Text.substr ( 16 );

  cout « "’" « Sub1 « "’" « endl;
  cout « "’" « Sub2 « "’" « endl;
}
```

Körning av programmet ger följande utskrift:

```
% ./Substring
’Rings’
’of’
%
```

18.9. Streams och strängar

Det finns några problem som relativt ofta förekommer då man gör program som hanterar strängar i olika former. Dessa är bl.a. hur man enkelt skapar strängar av olika andra primitiva datatyper och hur man läser data som finns i strängar och konverterar detta till någonting annat. Dessa båda problemen kan lösas med hjälp av *streams till/från stränger*.

Då vi hittills använt oss av streams har de alltid varit förknippade med skärmen, tangentbordet eller filer (se Kapitel 19), men aldrig med strängar. Det finns dock någonting som kallas *strängstreams* som vi kan använda för att läsa och skriva strängar.

18.9.1. Skriva till strängar

Vi kan tänka oss ett program som hanterar en SQL-databas och behöver skapa SQL-satser som används för att manipulera data i databasen. SQL-satser representeras normalt som strängar, och skickas sedan till databasen för exekvering, och eventuella resultat returneras på något databas-specifikt sätt. Vi kan tänka oss ett program som skall söka personer i ett företags personregister på bas av arbetsuppgift och lön. Vi vill skapa t.ex. följande SQL-sats:

```
SELECT *
FROM Persons
WHERE Job = 'Hacker' AND Salary > 20000;
```

Om man alltid vill exekvera exakt samma sats har vi inga problem, det är bara att skriva SQL-satsen i en sträng och saken är klar. Vill vi dock kunna ändra på värdena på `Job` och `Salary` under programmets körning måste vi skapa SQL-satsen dynamiskt. Här kan vi använda oss av klassen `ostrstream`. Denna tillåter att vi skriver data till en instans av klassen med « precis som om vi skrev ut data till skärmen. Sedan kan man "plocka ut" den resulterade strängen med metoden `str()`. Ett exempel på hur ovannämda SQL-sats kan konstrueras med hjälp av `ostrstream`:

```
#include <strstream>
#include <string>

int main () {
  int Salary;
  string Job;

  // läs in värden på lön och jobb
  cout « "Ge jobbtyp: ";
  cin » Job;
  cout « "Ge lön:     ";
  cin » Salary;

  // skapa SQL-satsen
  ostrstream SQL;
```

```
  SQL « "SELECT *" « "FROM PERSONS " « endl
      « "WHERE Job = '" « Job « "' AND Salary > "
      « Salary « ";";

  // skriv ut den nya satsen
  cout « endl « "SQL-satsen är nu: " « endl
       « SQL.str () « endl;

}
```

Vi skapar variabeln `SQL` som en instans av `ostrstream`. Sedan skriver vi vår önskade sträng precis som om vi skrev till `cout`. Vi kan skriva ut strängar, tal, tecken o.s.v., `ostrstream` kan konvertera alla datatyper till lämpliga strängar.

För att sedan få tillgång till den resulterade strängen använder vi metoden `str()` som returnerar en sträng, samt skriver ut denna. I ett större program skulle strängen t.ex. skickas till en databas för exekvering. Kör vi programmet ovan får vi t.ex. följande utskrift:

```
% ./StringStream1
Ge jobbtyp: Hacker
Ge lön:     20000

SQL-satsen är nu:
SELECT *FROM PERSONS
WHERE Job = 'Hacker' AND Salary > 20000;
%
```

Notera att detta är ett enkelt sätt att konvertera tal till motsvarande sträng! Ett exempel:

```
#include <strstream>
#include <string>

int main () {
  float Tal;

  // läs in ett tal
  cout « "Ge flyttal: ";
  cin » Tal;

  // skapa stream
  ostrstream Stream;

  // utför konverteringen
  Stream « Tal;
```

```
  // hämta det konverterade talet
  string TalStrang = Stream.str ();

  cout « "Talet som sträng: " « TalStrang « endl;
}
```

Programmets utskrift kan se ut på följande sätt:

```
% ./StringStream2
Ge flyttal: 3.045
Talet som sträng: 3.045
%
```

Notera att den headerfil som inkluderades är `<strstream>`, men enligt standarden för C++ borde filen heta `<sstream>`. Detta är ett resultat av att inte alla system ännu fullt följer standarden. Se Avsnitt 18.9.3 för mera information.

18.9.2. Läsa från strängar

Motsatsen till det ovannämda kan även göras, d.v.s. man kan läsa data ur en sträng precis på samma sätt som man kan läsa från `cin`. Detta gör det lättare att t.ex. skriva en parser som tolkar data ur en sträng. Klassen som används heter då `istrstream`. Vi kan skapa ett enkelt program som läser en rad med text och delar upp den i dess beståndsdelar, d.v.s. ord, på följande sätt:

```
#include <strstream>
#include <string>

int main () {
  string Rad;
  string Ord;
  int Antal = 1;

  // läs in en rad
  cout « "Ge en rad med text: ";
  getline ( cin, Rad );

  // skapa stream
  istrstream Stream ( Rad.c_str () );

  // iterera så länge det finns flera ord
  while ( Stream » Ord ) {
```

```
    cout « "Ord " « Antal++ « " = " « Ord « endl;
  }
}
```

Vi använder oss här av funktionen `getline()` för att läsa en hel rad med data från `cin` in i `Rad`. Mera information om denna funktion finns i Avsnitt 19.2.3. Normala `cin » Rad;` skulle inte gett önskat resultat, eftersom den endast läser ett ord åt gången. Därefter skapas en `istrstream` av den inlästa raden. Ett problem här är att den konstruktor som finns tillgänglig på det använda systemet inte accepterar `string`, utan kräver C-strängar, varvid metoden `c_str()` används för att konvertera till en sådan. Läs mera på Avsnitt 18.11. Efter detta är det sedan bara att läsa ord för ord ur den inlästa strängen så länge det finns data.

Denna klass kan även användas för att läsa andra datatyper än endast `string`. Vi kan således enkelt kontrollera om en inläst sträng är t.ex. ett giltigt tal eller inte med följande:

```
#include <strstream>
#include <string>

int main () {
  string Text;
  string Rest;
  int Tal;

  // läs in en rad
  cout « "Ge ett tal: ";
  cin » Text;

  // skapa stream
  istrstream Stream ( Text.c_str () );

  // läs ett tal om det går
  if ( Stream » Tal ) {
    // talet är ok
    cout « "Läste talet " « Tal « endl;

    // finns det mera text kvar i buffern?
    if ( Stream » Rest ) {
      // jep, mera text kvar, så det är kanske inte ett ok tal?
      cout « "Texten '" « Rest « "' blev oläst." « endl;
    }
  }
  else {
    // inget giltigt tal
    cout « "Texten '" « Text « "' är inte ett tal." « endl;
  }
```

```
}
```

Programmet läser ett tal och kontrollerar ifall det kunde konverteras till ett heltal. Notera dock att `istrstream` endast läser tecken ur strängen så länge som de passar in med den datatyp som läses. Om vi ger som tal texten `0xffff` i tron att den läser ett hexadecimalt tal kommer den endast att läsa den första nollan och ignorera resten. Därför har vi en extra `if`-sats för att kontrollera ifall det blev text kvar i strängen. Om det finns text kvar kunde inte hela ordet konverteras och det är troligtvis felaktigt. Samma händer om vi t.ex. försöker läsa strängen `10.73` in i ett heltal; allting efter nollan ignoreras. Körning av programmet kan t.ex. ge:

```
% ./StringStream4
Ge ett tal: 54
Läste talet 54
% ./StringStream4
Ge ett tal: 0xffff
Läste talet 0
Texten ’xffff’ blev oläst.
%
```

På detta sätt kan man bygga upp robust inläsning av data från tangentbord och filer. Det lönar sig alltid att kontrollera giltighet på inläst data!

18.9.3. Strängstreams och kompatibilitet

Som du märkt är det ännu vissa problem med system som inte fullt följer standarden för C++. Detta manifesteras ganska tydligt då man använder sig av sträng-streams. De klasser som används i exemplen ovan, d.v.s. `istrstream` och `ostrstream` heter egentligen `istringstream` och `ostringstream`, men dessa klasser finns inte på det system som denna kurs i huvudsak använder sig av. Analogt så skall man egengligen inkludera headerfilen <sstream≶ istället för <strstream> som vi gjort i exemplen. Om det system du använder följer standarden bättre skall du använda dessa filer och klasser istället.

I en senare version av detta kompendium kommer de riktiga klasserna och headerfilerna att användas.

18.10. Manipulera tecken

Många program behöver manipulera tecken i strängar direkt för att utföra någon viss operation. Man kan accessera tecknen enkelt via operatorn `[]` eller metoden `at()` och på så vis direkt använda de `char`:s som stängen är uååbyggd av. Det är dock arbetsdrygt och ganska svårt att på bas en `char` veta vad det är för *typ* av tecken. Program kan t.ex. behöva kontrollera om ett tecken är en siffra, en bokstav, whitespace o.s.v. Skall man göra detta manuellt varje gång det behövs blir program mycket svårare att skriva. Det är även skillnad på vilka tecken som kan räknas som alfanumeriska, beroende på vilken *locale* som används. En locale bestämmer olika aspekter på ett systems inställningar, såsom tid, teckentabeller o.s.v. Våra normala skandinaviska tecken å, ä, ö, Å, Ä och Ö är inte bokstäver i alla locales.

Man kan använda sig av en mängd funktioner som finns definierade i headefilen `ctype.h` för att känna igen och klassificera olika tecken. De olika funktionerna är:

- `isalpha()` kontrollerar om ett tecken är en bokstav i den använda locale:n.
- `isupper()` samma som `isalpha()`, men tecknet skall även vara en gemen (liten bokstav).
- `islower()` samma som `isalpha()`, men tecknet skall även vara en versal (stor bokstav).
- `isdigit()` kontrollerar om ett tecken är en siffra (0-9).
- `isxdigit()` kontrollerar om ett hexadecimalt tecken är en siffra (0-9, a-f, A-F).
- `isspace()` kontrollerar om ett tecken är whitespace (space, radbyte `\n`, tabulator `\t` eller vertikal tabulator `\v`).
- `iscntrl()` kontrollerar om ett tecken är ett kontrolltecken.

- `ispunct()` kontrollerar om ett tecken är ett skiljetecken.
- `isalnum()` kontrollerar om ett tecken är alfanumeriskt, d.v.s. antingen `isalpha()` eller `isdigit()`.
- `isprint()` kontrollerar om ett tecken kan skrivas ut (inklusive space).
- `isgraph()` kontrollerar om ett tecken kan skrivas ut (exklusive space).

Alla dessa metoder tar som parameter en `int`, men man kan förstås även skicka en `char`. Alla returnerar en `int` med värdet 0 då det givna teckent inte uppfyllde kriterierna och annars ett annat värde. Vi kan göra ett enkelt program för att kontrollera om en given sträng kan vara ett giltigt socialskyddssignum:

```
#include <iostream>
#include <string>
#include <ctype.h>
#include <stdlib.h>

bool okSignum (const string & Data) {
  // är längde ok?
  if ( Data.length () != 11 )
// kan inte vara ett signum
return false;

  // först skall vi ha 6 siffror
  for ( int Index = 0; Index < 6; Index++ ) {
if ( ! isdigit ( Data[Index] ) ) {
  // ingen siffra
  return false;
}
  }

  // nästa skall vara ett '-'
  if ( Data[6] != '-' )
    // inget '-'
    return false;

  // sedan 4 alfanumeriska tecken
  for ( int Index = 7; Index < 11; Index++ ) {
    if ( ! isalnum ( Data[Index] ) ) {
      // ingen siffra
      return false;
    }
  }

  // det kan vara ett signum
  return true;
}
```

```
int main (int argc, char * argv[]) {
  // verifiera antal parametrar
  if ( argc != 2 ) {
    cout « "Fel antal parametrar!" « endl;
    cout « "Användning: " « argv[0] « " signum" « endl;
    return EXIT_FAILURE;
  }

  string Signum = argv[1];

  // kolla ifall det är ett signum
  if ( okSignum ( Signum ) ) {
    cout « Signum « " kan vara ett giltigt socialskyddssignum." « endl;
  }
  else {
    cout « Signum « " är inte ett giltigt socialskyddssignum." « endl;
  }
}
```

Programmet kontrollerar inte att födelsedatumen är giltig, ej heller att slutdelen är giltig. För det finns det speciella algoritmer som faller utanför detta kompendiums omfång.

18.10.1. Konvertera tecken

Två speciella funktioner kan användas för att konvertera tecken till versaler respektive gemener. Dessa funktioner finns definierade i headefilen `ctype.h` och heter `toupper()` och `tolower()`. Båda tar som argument en `int` (ett tecken) och returnerar det konverterade tecknet. Ifall tecknet inte kunde konverteras returneras tecknet oförändrat. Vi kan göra två funktioner för att konvertera alla tecken i en sträng till versaler respektive gemener:

```
#include <iostream>
#include <string>
#include <ctype.h>
#include <stdlib.h>

string toUpper (const string & Data) {
  string Resultat;

  // reservera korrekt antal tecken
  Resultat.resize ( Data.length () );

  // iterera över hela strängen
  for ( unsigned int Index = 0; Index < Data.length (); Index++ ) {
```

```
Resultat[Index] = toupper ( Data[Index] );
  }

  return Resultat;
}

string toLower (const string & Data) {
  string Resultat;

  // reservera korrekt antal tecken
  Resultat.resize ( Data.length () );

  // iterera över hela strängen
  for ( unsigned int Index = 0; Index < Data.length (); Index++ ) {
Resultat[Index] = tolower ( Data[Index] );
  }

  return Resultat;
}
```

Vi kan sedan lägga till ett litet hivudprogram för att testa hur dessa funktioner fungerar:

```
int main (int argc, char * argv[]) {
  // verifiera antal parametrar
  if ( argc != 2 ) {
    cout « "Fel antal parametrar!" « endl;
    cout « "Användning: " « argv[0] « " signum" « endl;
    return EXIT_FAILURE;
  }

  string S = argv[1];

  // konvertera och skriv ut
  cout « "Som versaler: " « toUpper ( S ) « endl;
  cout « "Som gemener:  " « toLower ( S ) « endl;
}
```

Kör man programmet kan man t.ex. få följande resultat:

```
% ./Convert "Ett litet test, 123!"
Som versaler: ETT LITET TEST, 123!
Som gemener:  ett litet test, 123!
%
```

18.11. Konvertera till C-strängar

I vissa fall kan det vara nödvändigt att kunna konvertera en `string` till en C-sträng, t.ex. då man måste anropa en funktion eller metod som tar som parameter en C-sträng. Lösningen är då att använda någon av metoderna `data()` eller `c_str()`. Båda returnerar `char *`, med den skillnaden att den sträng som `c_str()` returnerar är terminerad av 0, medan den som `data()` returnerar inte är det. Strängar i C använder sig av en 0:a för att indikera strängens slut. Av detta följer att `c_str()` är den metod som bör användas i de flesta fall.

18.12. Iteratorer och `string`

Normala *STL-iteratorer* kan även användas tillsammans med `string` (se Kapitel 24 för mera information). Klassen `string` har metoderna `begin()`, `end()`, `rbegin()` och `rend()` som returnerar framåt- respektive bakåtiteratorer. Man kan därför enkelt iterera igenom en sträng med hjälp av dessa. Ett program som testar om en given sträng är ett palidrom kan skrivas med hjälp av iteratorer på följande sätt:

```
#include <iostream>
#include <string>

int main (int argc, char * argv[]) {

  // verifiera antal parametrar
  if ( argc != 2 ) {
cout « "Fel antal parametrar!" « endl;
cout « "Användning: " « argv[0] « " sträng" « endl;
return 1;
  }

  string S = argv[1];

  // iteratorer till första och sista tecknet
  string::iterator It1 = S.begin ();
  string::reverse_iterator It2 = S.rbegin ();
```

```
  for ( ; It1 != S.end () && It2 != S.rend (); It1++, It2++ ) {
// är tecknen lika?
if ( *It1 != *It2 ) {
  // tecknen olika, inget palindrom
  cout « S « " är inget palindrom" « endl;
  return 0;
}
  }

  // alla tecken lika, vi har ett palindrom
  cout « S « " är ett palindrom!" « endl;
}
```

Körning av programmet kan t.ex. ge resultatet:

```
% ./Palindrom test
test är inget palindrom
% ./Palindrom 1234321
1234321 är ett palindrom!
% ./Palindrom saippuakivikauppias
saippuakivikauppias är ett palindrom!
%
```

Kapitel 19. Filhantering

I detta kapitel redogör för hur man kan läsa och skriva data till olika filer i C++.

19.1. Klasser

Filer hanteras i C++ på samma sätt som man skriver data till skärmen via `cout` eller läser data från tangentbordet via `cin`. Dessa båda är instanser av klasserna `ostream` respektive `istream`. På samma sätt hanteras filer i C++ genom att skapa instanser av klasserna `ifstream` och `ofstream` för att läsa respektive skriva till en fil. Genom att alla former av I/O använder sig av liknande klasser kan man enkelt omvandla t.ex. en funktion att skriva till en fil istället för `cout`.

Notera att alla metoder som har presenteras även gäller för `cin`, `cout`, `cerr` och `clog`. Vi har hittills använt dem för att endast läsa och skriva primitiva datatyper, men de kan även hanteras på ett mera mångsidigt sätt.

19.2. Läsa från en fil

Att läsa från en fil är enkelt i C++. Exakt samma syntax används som då man läser från `cin`. Först måste man dock öppna en fil för läsning. Detta sker genom att skapa en instans av klassen `ifstream`. Denna tar som argument ett filnamn som skall vara en C-sträng (inte `string`). Allmänt görs det på föjande sätt:

```
#include <fstream>

ofstream namn (filnamn);
```

Alla filrelaterade streamklasser finns definierade i headerfilen `fstream`. Om man vill initiera en `ifstream` med ett filnamn som finns i en `string` man man använda dess metod `c_str()` för att få en C-sträng som kan användas. En alternativ konstruktor

finns som tar som andra argument en *bitmask* som representerar önskad funktionalitet hos filen:

```
ofstream namn (filnamn, openmode mask);
```

De olika konstanter som kan or:as ihop är definierade i klassen `ios`. De värden som påverkar `ifstream` är följande:

- `in` öppnar filen för läsning. Detta är standardvärdet ifall man skapar en `ifstream` med endast en parameter.
- `binary` specificerar att data skall läsas i binär form istället för text.

Man bör alltid kolla ifall filen kunde öppnas innan man försöker läsa data från den. Ifall filen inte existerar eller programmet inte har rätt att läsa filen kommer objektet inte att kunna användas för läsning. Man kan kolla detta med metoden `good()` eller med den överlagrade operatorn `!` enligt följande:

```
ifstream Fil ( "filnamn");

// använd antingen:
if ( ! Fil ) {
  // kunde inte öppna filen
}

// eller:
if ( ! Fil.good () ) {
  // kunde inte öppna filen
}
```

Vilken man använder har ingen betydelse.

Vi kan nu skapa ett enkelt program som öppnar en fil för läsning och läser samt räknar alla ord som finns i filen:

```
#include <fstream>
#include <string>

int main (int argc, char * argv[]) {
  int Antal = 0;
  string Ord;
```

```
  // har vi tillräckligt parametrar?
  if ( argc != 2 ) {
    cout « "Fel antal parametrar!" « endl;
    cout « "Användning: " « argv[0] « " filnamn" « endl;
    exit ( EXIT_FAILURE );
  }

  // öppna en fil
  ifstream Data ( argv[1] );
  if ( ! Data ) {
    // kunde inte öppna filen
    cout « "Kunde inte öppna filen: " « argv[1] « endl;
    exit ( EXIT_FAILURE );
  }

  // iterera över all ord i filen
  while ( Data » Ord ) {
    // öka antalet lästa ord
    Antal++;
  }

  // skriv ett svar
  cout « "Läste " « Antal « " ord." « endl;
}
```

Notera att vi inkluderar headerfilen `fstream`, och inte `iostream` som normalt. Data kan läsas från en `ifstream` exakt på samma sätt som från `cin`, d.v.s. alla primitiva datatyper kan läsas så väl som strängar.

19.2.1. Avsluta inläsning

Inläsning avslutas vanligen då man läst in den mängd data som behövs eller då slutet på filen nåtts. Den första möjligheten är lätt att hantera, eftersom programmet då vet hur den inlästa filen ser ut och vad som kan förväntas. Om programmet däremot inte vet någonting om den lästa filen vill man kanske läsa till till filens slut, d.v.s. *end-of-file*. Man kan kolla om en `ifstream` (och även en `istream` som t.ex. `cin`) har nått slutet på filen med metoden `eof()`. Denna returnerar `true` då slutet nåtts och `false` ifall mera data kan läsas. Programmet ovan använder sig av funktionaliteten att alla stream-klasser alltid returnerar sig själv vid läsning och skrivning med » respektive «, och en misslyckad läsning returnerar 0.

```
// iterera över all ord i filen
```

```
while ( Data » Ord ) {
  // öka antalet lästa ord
  Antal++;
}
```

Så stream:en `Data` returneras efter varje »-operation, och den sista operationen som misslyckas returnerar 0 eller `false`. För mera information om den överlagrade operatorn fungerar se Kapitel 22 och Kapitel 21. Vi kunde skriva om den del av koden som sköter inläsningen i programmet ovan till detta:

```
// iterera över all ord i filen
while ( ! Data.eof () ) {
  // läs ett ord
  if ( Data » Ord ) {
    // öka antalet lästa ord
    Antal++;
  }
}
```

Varför måste vi ha en `if`-sats runt inläsningen? Om vi inte har denna där kommer även den sista inläsningen som ju läser något som inte existerar, att även räknas med. Flaggan `eof()` sätts inte förrän *efter* att någon operation läst data som inte finns. Klassen kan inte veta ifall det finns mera data eller inte förrän någonting försöker läsa något som inte finns. På detta sätt räknar vi med endast de ord som vi lyckades läsa.

Ett alternativt sätt att åstadkomma samma funktionalitet är att använda `fail()` istället för `eof()` i koden ovan. Denna metod anger ifall nästa operation kommer att misslyckas eller inte. Den kan dock inte heller vet i förväg om nästa läsning kommer till filens slut, utan den kan endast på bas av status hos objektet i anropsögonblicket veta ifall nästa operation kommer att misslyckas.

19.2.2. Läsa teckenvis

Man kan förutom att läsa data från en `ifstream` (eller t.ex. `cin`) även läsa data tecken för tecken, eller ett antal tecken på en gång. Beroende på tillämpningen kan detta vara vad som behövs. För att läsa tecken för tecken används metoden `get()` med olika argument. Den enklaste formen används på följande sätt:

```
char Tecken
```

```
enFil.get ( Tecken );
```

Metoden `get()` läser alla tecken som finns i filen, även olika former av *whitespace* (mellanslag, tabulatorer m.m.), medan de olika formerna av » hoppar över whitespace. man bör sålunda använda t.ex. `get()` då man vill åt *exakt* det som finns i en fil.

Andra versioner av `get()` tar som parametrar en C-sträng, d.v.s. en `char *`. Med dessa kan man läsa ett antal tecken upp till ett givet maximum eller tills ett givet termineringstecken påträffas. Dessa ser ut på följande sätt:

```
char Buffer [storlek];
enFil.get ( Buffer, storlek );
// eller
enFil.get ( Buffer, storlek, terminator );
```

Dessa metoder läser maximalt `storlek - 1` tecken in i den buffer som ges med som parameter. Denna måste förstås ha utrymme för alla dessa tecken. Det sista teckent sätts alltid till 0 för att terminera strängen. Så om storleken ges till 10 läses endast maximalt 9 tecken in medan det tionde sätts till `'\0'`. Metoden läser i den första utformningen ända tills ett radbytestecken (`'\n'`) hittas. I den andra formen kan man själv specidicera vilket tecken som skall vara det terminerande. Vill man läsa rader är `'\n'` lämpligt, men vill man t.ex. läsa ord kan man separera på ett mellanslag istället med `' '`.

Dessa två varianter av `get()` har ett problem. De läser inte bort det tecken som terminerade inläsningen. Om vi t.ex. terminerar på ett `'\n'` läser `get()` inte bort teckent, utan det finns kvar då nästa läsning utförs. Vi måste manuellt läsa bort tecknet, t.ex. med en annan `get()`. En aning arbetsdrygt. Vi kan göra ett program som räknar antalet rader i ett program (en form av programmet `wc`) enligt följande:

```
#include <fstream>

int main (int argc, char * argv[]) {
  int Antal = 0;
  char Buffer [250];

  // kontrollera argument till main()
  ...

  // öppna en fil, samma
  ifstream Data ( argv[1] );
```

```
  // felhantering från exemplen ovan
  ...

  // läs rad för rad
  while ( Data.get ( Buffer, 250 ) ) {
    // läs det '\n' som finns
    Data.get (Newline);

    // öka antalet lästa tecken
    Antal++;
  }

  // skriv ett svar
  cout « "Läste " « Antal « " tecken." « endl;
}
```

Vissa delar är bortlämnade, men de är identiska med programmen ovan. Notera att vi har en extra

```
Data.get (Newline);
```

för att läsa bort det radbytestecken som terminerade inläsningen. Programmet antar att de inlästa raderna är under 250 tecken långa, annars utförs flera läsningar på samma rad och raden räknas multipelt med i antalet rader och vi får ett felaktigt resultat.

En sista form av att läsa teckenvis är med metoden `read()`. Den skiljer sig från `get()` i och med att den int har något speciellt tecken som terminerar inläsningen, utan det givna antalet tecken läses alltid in. Den terminerar dock då slutet på filen nåtts. Metoden placerar inget `'\0'` som det sista tecknet, utan den läser hela den givna buffern full med data (om så mycket data kunde läsas). Använd denna endast om du vet vad du gör! Kan användas för binär läsning (Avsnitt 19.2.5.

19.2.3. Läsa radvis

Ett alternativ till att läsa filer teckenvis är att läsa radvis. Detta utförs med metoden `getline()` som även den finns i flera olika varianter. De två normala varianterna fungerar exakt såsom de olika varianterna av metoden `get()` (se Avsnitt 19.2.2), med den signifikanta skillnaden att `getline()` även läser bort det terminerande tecknet. Vi kan således skriva om huvudslinga för vårt ordräkningsprogram på följande sätt:

```
char Buffer [250];
....

// läs rad för rad
while ( Data.getline ( Buffer, 250 ) ) {
  // öka antalet lästa rader
  Antal++;
}
```

Vi måste inte läsa bort ett `'\n'` med en extra `get()`, utan den läses av `getline()`. Vi kan även använda den andra formen av metoden `getline()` för specificera ett annat termineringstecken än `'\n'`.

För klassen `string` finns en extra *funktion* definierad som heter `getline()`. Notera ettd et inte är frågan om en metod! Denna funktion fungerar på samma sätt som `getline()` ovan, men man måste explicit ge med en `ifstream` eller `istream` som första parameter. Strängen ges som andra parameter. Vi behöver inte ge någon storlek för denna funktion, eftersom `string` växer vid behov. Detta är det mest robusta sättet att läsa en rad, men även det långsammaste. För det mesta lönar det sig dock att offra en aning hastighet och vinna robusthet. Vi kan skriva om slingan ovan till:

```
string Buffer;
...

// läs rad för rad
while ( getline ( Data, Buffer ) ) {
  //  while ( Data.getline ( Buffer ) ) {
  // öka antalet lästa rader
  Antal++;
}
```

Notera att vi använder en funktion och inte en metod. Funktionen `getline()` är definierad i filen `<string>`.

19.2.4. Ignorera tecken

Ibland kanske man vill kunna läsa tecken ur en fil eller annan in-stream och ignorera dem tills något specifikt tecken läses. Detta kan göras med metoden `ignore()` som tar som parametrar antalet tecken som maximalt skall ignoreras och ett termineringstecken

som avslutar ignoreringen. Kan användas t.ex. på följande sätt för att ignorera max 80 tecken eller tills nästa `;` läses:

```
enFil.ignore (80, ';');
```

De tecken som läses sparas ingenstans. Båda parametrarna har standardvärden som gör att `ignore()` utan parametrar ignorerar ett tecken och avslutar då filens slut nåtts. Vi kan skriva om programmet ovan som läser rader m.h.a. `get()` (se Avsnitt 19.2.2) till följande:

```
// läs rad för rad
while ( Data.get ( Buffer, 250 ) ) {
  // ignorera det '\n' som finns
  Data.ignore ();

  // öka antalet lästa rader
  Antal++;
}
```

Slingan blir en aning lättare att förstå.

19.2.5. Läsa binär data

Man kan även med `ifstream` läsa data som är i *binär* form, d.v.s. inte ASCII. Filer i binär form kan sällan läsas av andra program än det program som skapat filen, d.v.s. man kan inte titta meningsfullt på filens innehåll med t.ex. `more` eller `notepad`, utan filen verkar innehålla en massa skräp. Om vi t.ex. skriver ut talet 123456 (en `int`) till en textuell fil skrivs bokstäverna '1', '2', '3', '4', '5' och '6' till filen. Skriver vi däremot i binär form skrivs de fyra (eller annat antal) *bytes* ut som datatypen består av.

För att öppna en fil för läsning i binärt format använder man sig av flaggan `ios::binary` då man öppnar filen. T.ex. kan vi öppna en binär fil för läsning på följande sätt:

```
ifstream BinData ( "register.dat", ios::binary );
```

Notera att denna flagga inte har någon verkan under Unix! För att sedan läsa binär data från filen *borde* man kunna använda normala », men under Unix bör man använda sig

av metoden `read()`. Ett program som läser in en serie `int`:s från en fil kan skrivas på följande sätt:

```
#include <fstream>
#include <stdlib.h>

int main () {
  int Tmp;

  // öppna filen
  ifstream In ( "Test.bin");

  // kunde filen öppnas?
  if ( ! In ) {
// kunde inte öppna filen!
cout « "Kunde inte öppna filen 'Test.bin'!" « endl;
exit ( EXIT_FAILURE );
  }

  // läs in tal från filen
  while ( In.read ( &Tmp, sizeof(int) ) ) {
cout « "Läste: " « Tmp « endl;
  }
}
```

Vi använder `sizeof(int)` för att portabelt kunna ta reda på storleken på en `int`, så att `read()` kan läsa korrekt antal bytes. Portabilitet är då man använder binära filer ganska sekundärt, eftersom man ändå i många fall inte kan utbyta binära datafiler mellan olika system p.g.a. olika storlekar på datatyper. Om man inte har ett absolut behov av binär data bör man använda sig av datafiler i textform. Undantag är bl.a. fall där filer i textform blir för stora, eller där bandbredden är begränsad (t.ex. nätverk).

19.3. Skriva data till fil

Att skriva data till en fil är betydligt mycket enklare än att läsa data. Man kan se på det som att skriva till skärmen, men data skrivs istället till en fil.

Den klass som används för att skriva data är `ofstream`. Även denna finns definierad i `<stream>`. För att skriva till en fil kan vi t.ex. använda följande syntax:

```
#include <fstream>
```

```
#include <stdlib.h>

int main () {
  // öppna filen
  ofstream Ut ( "index.html");

  // kunde filen öppnas?
  if ( ! Ut ) {
    // kunde inte öppna filen!
    cout « "Kunde inte öppna filen 'index.html'!" « endl;
    exit ( EXIT_FAILURE );
  }

  // skriv en html-version av 'Hello world'
  Ut « "<html><head>" « endl;
  Ut « "<title>Hello world</title>" « endl;
  Ut « "<body>" « endl;
  Ut « "Hello world" « endl;
  Ut « "</body></html>" « endl;
}
```

Filen vi får ser ut på följande sätt:

```
% ./Write1
% cat index.html
<html><head>
<title>Hello world</title>
<body>
Hello world
</body></html>
```

Alla normala datatyper kan skrivas ut, helt på normalt sätt.

Det finns en del flaggor som kan användas då man öppnar filer för skrivning, precis som när man öppnar dem för läsning. Dessa flaggor finns definierade i `ios` och är:

- `app` öppnar filen för *appendering*, d.v.s. data skrivs till på slutet av filen, istället för som normalt skriva över en fil. Kan användas för t.ex. logfiler.
- `ate` öppnar filen och "söker" till slutet av filen.
- `binary` öppnar filen i binär form. Ingen inverkan under Unix.
- `out` öppnar en fil för skrivning. Standard för `ofstream`.

- `trunc` öppnar en fil för skrivning och *trunkerar* innehållet, d.v.s. skriver över allt innehåll. Standard för `ofstream`.

De vanligaste flaggorna man använder är `ios::app` för att skriva till slutet av en fil.

19.3.1. Skriva teckenvis

Precis som man kan läsa teckenvis kan man även skriva data teckenvis. Man kan då använda metoden `put()` för att skriva ut ett enda tecken. Denna används i följande program för att skriva ut alla programmets kommandoradsparametrar till en fil `params.txt`

```
#include <fstream>
#include <stdlib.h>
#include <string>

int main (int argc, char * argv[]) {
  // öppna filen
  ofstream Ut ( "params.txt" );

  // kunde filen öppnas?
  if ( ! Ut ) {
    // kunde inte öppna filen!
    cout « "Kunde inte öppna filen ’params.txt!" « endl;
    exit ( EXIT_FAILURE );
  }

  // iterera över alla parametrar vi har
  for ( int Index = 0; Index < argc; Index++ ) {
// hämta parameter
string Parameter = argv[Index];

// iterera över alla tecken i strängen
for ( unsigned int Tecken = 0; Tecken < Parameter.length (); Tecken++ ) {
  // skriv ut ett tecken
  Ut.put ( Parameter[Tecken] );
}

// skriv ett radbyte
Ut.put ( ’\n’ );
  }
}
```

Vi indexerar här strängen `Parameter` tecken för tecken och skriver ut dem. Vi kunde lika väl ha direkt skrivit ut strängen med hjälp av «.

19.3.2. Skriva binär data

Man kan läsa binär data i C++, så följdaktligen måste det finnas ett sätt att även skriva binär data. Man kan använda sig av metoden `read()` för att läsa binärt, så för att skriva används en metod som heter `write()`. Denna metod tar som argument en pekare och ett tal som anger antalet *bytes* som skall skrivas ut. Man kan här använda `sizeof()` ifall man inte är säker. Vi kan nu skriva ett program som skriver ut ett antal `int`:s i binär form. Denna data kan läsas av programmet i Avsnitt 19.2.5.

```
#include <fstream>
#include <stdlib.h>

int main () {
  // öppna filen
  ofstream Out ( "Test.bin" );

  // kunde filen öppnas?
  if ( ! Out ) {
    // kunde inte öppna filen!
    cout « "Kunde inte öppna filen 'Test.bin'!" « endl;
    exit ( EXIT_FAILURE );
  }

  // skriv ut 10 tal till filen
  for ( int Index = 0; Index < 10; Index++ ) {
    // skriv ut talet binärt
    Out.write ( &Index, sizeof(int) );
  }
}
```

Notera att vi inte använder flaggan `ios::binary` i detta exempel, eftersom flaggan inte har någon verkan under Unix. Under Windows (och andra system) kanske flaggan är nödvändig.

Som tidigare nämnts (se Avsnitt 19.2.5) bör man inte använda binära filer om inte omständigheterna kräver det.

19.4. Hantera felsituationer

Felsituationer händer ofta vid input från filer eller tangentbordet. Man kan aldrig vara säker på vad användaren skriver in eller vad som kan finnas i en fil som läses. Programmet måste helt enkelt vara förberett på att hantera fel av olika slag och kunna fortsätta exekveringen trots att inläst data inte på alla sätt uppfyller kriterierna för "korrekt data". Varje operation som vi använt tidigare i detta kapitel returnerar 0 ifall läsningen misslyckades, så vi kan använda oss av detta för att kontrollera att vi läser giltig data. Vi kan även använda oss av metoden `fail()` för att kontrollera om nästa läsning kommer att misslyckas. Problem får vi då vi skall bestämma oss för vad som skall göras med det felaktiga data som finns. Skall man läsa in och ignorera detta, och hur långt skall man i så fall ignorera input?

Vi kan skriva ett rudimentärt program som hanterar inläsning av heltal från tangentbordet på följande sätt:

```
#include <iostream>
#include <string>

int main () {
  int Tal;
  string Dummy;

  // iterera evigt
  while ( true ) {
    // läs ett tal
    cout « "Ge ett heltal: " « flush;

// läs ett tal
    if ( ! (cin » Tal) ) {
      // illegalt tal
      cout « "Illegalt heltal!" « endl;

      // nollställ felindikatorn för cin
      cin.clear ();

      // läs bort resten av raden och ignorera denna
      getline ( cin, Dummy );
    }
    else {
      // talet är ok
      cout « Tal « " är ett ok heltal!" « endl;
    }
  }
}
```

Notera parentesen run `cin » Tal`. Utan denna kommer `!` att evalueras före » tack vare högre precedens, och vi får ett resultat som inte är vad vi avser. Vi använder oss här av

metoden `clear()` som nollställer alla felindikatorer hos `cin` för att vi skall kunna fortsätta läsa bort de illegala tecknen. Därefter läses alla tecken t.o.m. ett radbytestecken in i `Dummy` och ignoreras. På detta sätt får vi en rudimentär hantering av fel. Man nolställer alltså felflaggor med `clear()` enligt:

```
enFil.clear ();
```

Det finns även andra metoder som kan användas för `ifstream` och `istream`, nämligen `good()` och `bad()`. Den förra anger att allting är i sin ordning och att nästa inläsning kan lyckas, medan `bad()` indikerar att någonting är allvarligt på tok och att inget kan sägas om vad som finns kvar att läsa. Skillnaden mellan `fail()` och `bad()` är att den förra indikerar ett fel i inläsningen men att inga tecken förlorats och man kan fortsätta läsa, medan den senare indikerar ett alvarligare fel som man kanske inte kan återhämta sig ifrån.

19.5. Stänga en fil

Filer stängs automatiskt av destruktorerna för de olika stream-klasserna, men de kan även explicit stängas med metoden `close()`. Denna tar inga parametrar och stänger filen. Kan användas enligt följande:

```
ifstream Fil ( "filnamn");

// använd antingen:
if ( ! Fil ) {
  // kunde inte öppna filen
}

// läs data
Fil » ...

// stäng filen
Fil.close ();
```

Kapitel 20. Exceptions

Detta kapitel behandlar det avancerade felhanteringssystemet i C++ som kallas *exceptions* (undantag). Exceptions tillåter mycket flexibel hantering och rapportering av fel.

20.1. Vad är *exceptions*

Ett programs robusthet mäts bra på hur väl det hanterar och återhämtar sig från olika felsituationer. I ett idealiskt program skall ett fel hanteras på korrekt sätt och sedan skall exekveringen fortsätta så gott det går. Det är dock ganska svårt att i ett stort program förmedla felsituationer mellan olika delar av programmet. Traditionellt i C och många andra språk har man förmedlat felmeddelanden i ett program antingen genom vissa överenskommna returnvärden från funktioner eller genom att helt enkelt avsluta programmet. Den första modellen har sina fördelar och nackdelar. I följande funktion fungerar denna helt ok:

```
Image * createImage (string Filename) {
  // försök läsa filen
  if ( ! readFile ( Filename ) {
    // kunde inte läsa filem
    return 0;
  }

  // returnera en ny Image
  return new Image ( ... );
}

int main () {
  string File = ....;

  // försök läsa en fil
  if ( ( NewImage = createImage (File ) ) == 0 ) {
    // kunde inte läsa filen, hantera felet
    ....
  }
  ...
}
```

Programmet ovan försöker läsa en fil och skapa en `Image` på bas av läst data. Om filen inte kunde läsas returnerar funktionen värdet 0. Detta värde är enkelt att jämföra med, eftersom 0 aldrig är en adress till ett giltigt objekt. Vi har i detta fall ett bra returvärde som indikerar fel. I följande exempel är det inte lika enkelt längre:

```
float divide (float X, float Y) {
  // dividerar vi med 0?
  if ( Y == 0 ) {
    // jep, vad ska vi göra nu? Vilket värde skall returneras?
    cout « "Division med 0" « endl;
  }
  else {
    return X / Y;
  }
}
```

Funktionen `divide()` får problem om användaren försöker dividera med 0, vilket inte är tillåtet. Funktionen kan skriva ut ett felmeddelande på skärmen, men det kanske inte ses av användaren, eller så kan den terminera programmet, vilket inte heller är en bra ide. Det finns inte heller något returvärde som kunde indikera att ett fel har skett. Problemet blir svårare om ett fel hnder djupt nästlat i funktionsanropen, och koden som kan hantera felet är flera nivåer uppåt i anropshierarkin. Programmet måste då på något meningsfullt sätt förmedla information om felet till felhanteraren.

I sammanhang som dess är C++ *exceptions* väldigt praktiska. De är C++:s sätt att förmedla olika typer av fel. Med exceptions slipper man bry sig om dylika problem och har större frihet att utnyttja t.ex. returvärden till "nyttig" information och inte felmeddelanden. Exceptions kan även förmedla mera information, såsom t.ex. en sträng som innehåller en text som kan skrivas ut på skärmen samt annan information om exakt vad som gick fel.

20.2. Använda exceptions

Det finns två skilda moment involverade då man använder sig av exceptions, en part som försöker exekvera en viss del av programmet och fånga upp fel, och en ennan part

som kan ge en exception. Då en exception aktiveras så *kastar* (throw) programmet en exception. Mottagaren har en skild sektion som är markerad som kritisk och exceptions som kastas från denna kritiska sektion fångas upp av en fångst-sektion. Vi ska se på ett konkret exempel på ett program som dividerar två tal som användaren matat in och som kastar en exception om man försöker dividera med 0:

```
#include <iostream>
#include <string>

float divide (float X, float Y) {
  // är täljaren 0?
  if ( Y == 0 ) {
    // jep, kasta en exception
    throw string ("Nämnaren måste vara != 0");
  }

  // värdena ok, räkna och returnera
  return X / Y;
}

int main () {
  float X, Y;

  // evig slinga
  while ( 1 ) {
    cout « "Ge täljaren: ";
    cin » X;
    cout « "Ge nämnaren: ";
    cin » Y;

    // försök skriva ut kvoten och fånga fel
    try {
      // kritisk sektion
      cout « X « "/" « Y « " = " « divide (X, Y) « endl;
    }
    catch (string Error) {
      // ett fel har fångats, skriv ut meddelandet
      cout « Error « endl;
    }
  }
}
```

Funktionen `divide()` kollar om nämnaren är 0, och i så fall kastas en exception. Detta görs med nyckelordet `throw`. Man kan kasta vilken datatyp som helst, allt från `int`, `string` till skräddarsydda klasser. I vårt fall villvi endast förmedla ett meddelande om att vi dividerar med 0. anroparen vet även att detta är det enda fel som kan uppstå. Ifall flera olika fel kunde uppstå i våra beräkningar kunde vi kasta t.ex. en `enum` och sedan i

anroparen kolla vilket fel som inträffade. I `main()` finns en slinga som läser in två tal och försöker dividera dessa. Den sektion där programmet förväntar sig en exception är i ett s.k. `try`-block. Efter detta kommer att s.k. `catch`-block där man deklarerar de exceptions som man är förberedd på att ta emot. I programmet ovan tas en `string` emot. Man kan ha flera `catch`-block för att ta emot olika typer av exceptions. Allmänt ser en exception-deklaration ut på följande sätt:

```
try {
  satser;
}
catch (datatyp1 variabel1) {
  satser;
}
catch (datatyp2 variabel2) {
  satser;
}
...
catch (...) {
  satser;
}
```

De olika `catch`-blocken måste ha olika datatyp som de försöker fånga. Man kan placera en sista "fånga alla" `catch` med datatypen `...` (tre punkter) för att fånga upp exceptions som inte tidigare fångats. Ifall där man har klasser som exceptions kommer det kastade undantaget att provas uppifrån och ned bland `catch`-satserna tills någon passar eller man når en `...`-sats. Det är även möjligt att nästa exceptions, d.v.s. ha ett nytt `try-catch`-block innanför t.ex. en catch-sektion.

20.2.1. Allokering och exceptions

Ett exempel på en operator som använder sig av exceptions är `new`, som kastar `bad_alloc` ifall den inte kunde allokera önskad mängd minne. Följande program allokerar minne tills det tar slut (ett bra exempel på faran med att läcka minne!):

```
#include <iostream>
#include <new>
```

```
int main () {
  int Allokerat = 0;

  while ( 1 ) {
    // försök allokera minne evigt
    try {
      new char [ 1000000 ];

      // åter en MB alokerat
      Allokerat++;
    }
    catch ( bad_alloc ) {
      // minnet är slut
      cout « "Minnet slut efter " « Allokerat « "Mb" « endl;
      return 1;
    }
  }
}
```

Allokeringen med new allokerar 1Mb med minne per gång och läcker det. Förr eller senare tar minnet slut i datorn och new misslyckas. Den kastar då bad_alloc. För att kunna använda denna exception måste man inkluderar <new>, som innehåller definitionen på bad_alloc.

20.2.2. Exceptions och konstruktorer

Man kan även använda exceptions tillsammans med konstruktorer. Tittar man närmare på en konstruktor märker man att den inte kan returnera något till anroparen. En konstruktor har inget sätt med vilket den kan förmedla att den inte kunde exekveras korrekt. Det finns flera olika suboptimala lösningar på problemet men exceptions torde vara den bästa lösningen. Följande situation kunde uppstå för en klass File som abstraherar en fil då man försöker skapa en instans genom att ge et filnamn med:

```
try {
  // försök öppna filen
  File * InputData = new File ( Filename );

  // filen öppnades ok, läs data
  ...
}
catch ( FileNotFound E ) {
  // ingen sådan fil hittades
}
catch ( PermissionDenied E ) {
```

```
  // inga rättigheter att öppna/läsa filen
}
catch ( ... ) {
  // annat fel, kanske minnesallokering
}

// programmet fortsätter med filen läst eller felet hanterat
```

20.2.3. Återkasta en exception

Ett `catch`-block kan om det vill kasta vidare samma exception som det har fångat genom en tom `throw;` utan exception efter. På så vis skickas samma exception uppåt i anropshierarkin. Det gör det lätt för olika nivåer att reagera på ett felmeddelande utan större besvär. På så vis kan en `catch` på lägsta nivå försöka ordna upp felet innan den ger upp och skickar felet vidare till en högre nivå. Exemplet nedan visar hur man kan återkasta en exception.

```
#include <iostream>

// definiera en egen datatyp för exceptions
enum Exception { Problem, Illa, Hemsk };

void thrower () {
  // något illa har hänt!
  throw Illa;
}

void catcher () {
  // försök anropa thrower och se vad som händer
  try {
    thrower ();
  }
  catch ( Exception E ) {
    // är det något vi kan hanter
    if ( E == Problem ) {
      // allt är ok, vi kan hantera problemt
      cout « "Ingen fara!" « endl;
    }
    else {
      // allvarligt problem
      throw;
    }
  }
}

void dummy () {
```

```
  // ingen 'try' här inte
  catcher ();
}

int main () {
  // anropa 'dummy'
  try {
    dummy ();
  }
  catch ( Exception E ) {
    // vilken typ av exception är det frågan om?
    switch ( E ) {
    case Problem : cout « "Ett litet problem" « endl; break;
    case Illa :    cout « "Ett större problem" « endl; break;
    case Hemsk :   cout « "Ett hemskt problem" « endl; break;
    }
  }
}
```

Vi har funktionen `thrower()` som kastar en exception av typen `Exception`. Det första `catch`-blocket i `catcher()` tar emot undantaget och kontrollerar vilken typ av exception det är. Om det är en typ som den kan hantera (`Problem`) hanteras problemet internt, annars är det ett större problem och en tom `throw` återkastar undantaget vidare. I funktionen `dummy()` finns inget `try-catch`-block så exceptionen skickas vidare till `main()`. Man behöver alltså inte ha ett `try-catch`-block på varje nivå, utan man kan kasta en exception mellan många nivåer. I `main()` kollas sedan vilken typ av excaption det var frågan om och denna hanteras på något sätt.

Notera den egna datatypen `Exception` som definierats för våra felmeddelanden. Den innehåller ingen text el.dyl., men i detta exempel är det endast typen av exception som är viktig.

20.3. Ofångade exceptions

Vad händer med exceptions som inte fångas av något `catch`-block? Det korta svaret är att de terminerar programmet. Ett program som kastar en exception som inte fångas upp på någon nivå i programmet, inte ens i `main()`, kommer att anropa en inbyggd funktion `terminate()`, som i sin tur kallar på `abort()` för att terminera programmet. Funktionen `abort()` är en normal C-funktion som kan användas om headerfilen

`<stdlib.h>` inkluderas. Den kan användas t.ex. till att avsluta ett program ifall något kritiskt fel hänt. `abort()` genererar en coredump, d.v.s. en fil som heter `core`, och som kan utforskas med en debugger.

Vi kanske inte alltid vill att vårt program skall terminera direkt om vi har en ofångad exception. Vi kanske har någon viss resurs eller något i vårt program som måste stängas, firgöras eller på ett annat sätt hanteras. Vi kan i sådana fall specifierara en egen funktion som skall anropas istället för `terminate()`. Följande program illustrerar hur det kan se ut:

```
#include <iostream>
#include <exception>
#include <stdlib.h>

// vår egna termineringsfunktion
void betterTerminate () {
  // vår specieliserade felhantering...
  cout « "Aiiee..." « endl;

  // vi måste avsluta programmet
  abort ();
}

int main () {
  // sätt en ny 'teminate'
  set_terminate ( betterTerminate );

  cout « "Kastar en exception." « endl;

  // kasta en exception
  throw 1;
}
```

Notera att vi även inkluderat `<exception>` för att få tillgång till funktionen `set_terminate`. Denna funktion tar som argument en pekare till en funktion som returnerar `void` och inte behöver några parametrar. Varje funktion är en pekare is sig, så vi kan direkt ge namnet på vår egna termineringsfunktion `betterTerminate`. Vi för inte någon egentlig nytta i `betterTerminate`, endast ett meddelande skrivs ut på skärmen och däreftern anropas `abort()`. En termineringsfunktion returnerar aldrig, så den bör anropa t.ex. `exit()` eller `abort()` för att terminera programmet. Då programmet ovan körs genereras följande utskrift (programmet har kallats `Exception5`):

```
% Exception5
Kastar en exception.
Aiiee...
Abort (core dumped)
```

Det sista meddelandet kommer från `abort()`, och indikerar att en `core`-fil genererats. På icke-Unixsystem genererar abort troligtvis inte en `core`-fil. Nyttan med att använda egna termineringsfunktioner är relativt marginell. Det vore istället bättre att installera ordentliga excaption-hanterare som fångar alla typer av exceptions före termineringsfunktionen anropas, för att på så vis få en möjlighet att meningsfullt reagera på det som orsakade felet.

20.4. Hierarkier med exceptions

Många västrukturerade system definierar en egen klass för exceptions. Det är som tidigare nämnt helt tillåtet att kasta en klass som en exception. Vi skall definiera en enkel excaption-klass:

```
#include <string>

// exception-klass
class Exception {
public:
  Exception (string Text = "Exception") : m_Text (Text) { }
  string m_Text;
};
```

Vi har här en enkel exception som innehåller en sträng som fungerar som ett meddelande. Detta meddelanda kan t.ex. skrivas ut på skärmen eller skrivas till en loggfil. Texten som ges har det fördefinierade värdet `"Exception"`. Ganska meningslöst, men fyller sin funktion. Notera att vi automatiskt initialiserar `m_Text` med den givna texten (se Avsnitt 14.3.1). Vi kan nu använda denna klass precis som vilken exception som helst.

20.4.1. Ärvda exceptions

Att ha en enkel klass för att representera en exception är kanske inte så väldigt praktiskt. Vi kan förstås skicka med en sträng som berättar vad som hänt, mend et är svårt för ett `catch`-block att veta vad som gått fel. Det vore bra att kunna ha mera specificerade exceptions för att göra det lättare att skapa vettiga `catch`-block. Ingenting hidrar oss då ifrån att ärva subklasser av vår `Exception`-klass:

```
#include <iostream>
#include <string>

// exception-klass
class Exception {
public:
  Exception (string Text = "Exception") : m_Text (Text) { }
  string m_Text;
};

// subklasser
class FileNotFound : public Exception {
public:
  FileNotFound (string Text = "File not found") : Exception (Text) { }
};

class PermissionDenied : public Exception {
public:
  PermissionDenied (string Text = "Permission denied") : Exception (Text) { }
};

class InvalidData : public Exception {
public:
  InvalidData (string Text = "Invalid data") : Exception (Text) { }
};

// funktion som genererar en exception
void test () {
  throw FileNotFound ( string("ingen sådan fil"));
}

int main () {
  // försök köra 'test'
  try {
    test ();
  }
  catch ( PermissionDenied E ){
    cout « "Kunde inte öppna filen: " « E.m_Text « endl;
  }
  catch ( FileNotFound E ){
    cout « "Hittade inte filen: " « E.m_Text « endl;
  }
  catch ( InvalidData E ){
    cout « "Felaktig data: " « E.m_Text « endl;
  }
```

```
  catch ( Exception E ){
    cout « "Exception: " « E.m_Text « endl;
  }
  catch ( ... ){
    cout « "Annan exception" « endl;
  }
}
```

Exemplet använder sig av tre subklasser till `Exception`. De alla kan ha ett unikt meddelande, och det är enkelt för `main()` att kontrollera vilken typ av exception det är frågan om. Notera att en exception i `catch`-block prövas uppifrån och ned, så det första `catch`-block som passar kommer att användas, de övriga ignoreras. Så om vi t.ex. placerar det block som fångar `Exception E` först av alla kommer det att passa även alla subklasser av `Exception`, och därmed inte använda de specialiserade blocken längre ned. Som tumregel kan sägas att man bör placera de mest subklassade eller mest specialiserade blocken först, och gå mot basklasser längre ner. Har man `...` med bör den vara sist.

Kapitel 21. Överlagring av operatorer

Detta kapitel behandlar hur olika operatorer kan överlagras för att göra klasser mera transparenta för programmeraren.

21.1. Vad är *överlagring av operatorer*?

I ett tidigare kapitel behandlade vi hur funktioner och metoder kan överlagras med hjälp av *polymorfism* (se Avsnitt 17.3). Överlagring av metoder låter oss ha metoder med samma namn men som tar olika parametrar. På samma sätt kan man även överlagra *operatorer* i C++ och lägga till eller ändra funktionalitet. Vi kan på detta sätt definiera olika operatorer som annars inte kunde användas med någon viss klass att ha en meningsfull betydelse. Genom detta kan man göra olika klasser mera *transparenta* för användaren. De blir lättare och mera intuitiva att använda och påminner mera om primitiva datatyper såsom t.ex. `int` eller `double`.

21.1.1. Operatorer som kan överlagras

En operator i C++ är en "symbol" som opererar på en datatyp. Det mesta av den text som binder ihop datatyper är olika operatorer. De olika operatorer som kan överlagras i C++ är följande:

Tabell 21-1. Operatorer som kan överlagras

<table>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td><td>&</td></tr>
<tr><td>|</td><td>~</td><td>!</td><td>=</td><td><</td><td>></td><td>+=</td></tr>
<tr><td>-=</td><td>*=</td><td>/=</td><td>%=</td><td>^=</td><td>&=</td><td>|=</td></tr>
<tr><td>«</td><td>»</td><td>«=</td><td>»=</td><td>==</td><td>!=</td><td><=</td></tr>
<tr><td>>=</td><td>&&</td><td>||</td><td>++</td><td>--</td><td>->*</td><td>,</td></tr>
<tr><td>-></td><td>[]</td><td>()</td><td>new</td><td>new[]</td><td>delete</td><td>delete[]</td></tr>
</table>

Inga andra operatorer än de ovannämda kan överlagras, men de flesta som är användbara finns bland dessa. Man vill relativt sällan överlagra andra än normala räkneoperatorer, jämförelse- och tilldelningsoperatorerna. Avancerade klasser som vill ha egen minneshantering kan dock överlagra t.ex. `new` och `delete`, men det ämnet går utanför detta kompendiums omfång. Se kursboken för mera information.

Då operatorer överlagras måste de bibehålla samma antal parametrar som de normalt gör. Man kan sålunda inte definiera en divisionsoperator (`/`) som tar en eller tre operander (parametrar). Divisionsoperatorn skall ha två operander. Däremot kan t.ex. operatorn minus (`-`) mycket väl ha en eller två parametrar, d.v.s. normal subtraktion (två parametrar) och negation (en parameter). Man kan inte ändra operatorers precedensordning från vad den är i normala fall. Enda möjligheten är att använda parenteser.

Det är inte heller vettigt att ändra på en operators funktionalitet genom överlagring. Det är helt möjligt att överlagra tilldelningsoperatorn till att skriva ut ett objekt på skärmen, men det är inte vettigt. Användaren har en grunduppfattning om vad som borde hända då t.ex. `=` används, och man bör inte ändra betydelsen. Man skall bibehålla operatorers normala betydelse, och endast göra funktionaliteten bättre om den redan finns, och implementera den normala funktionaliteten om operatorn inte redan kan användas tillsammans med klassen i fråga.

21.1.2. Exempelklassen `Vector`

I detta kapitel skall vi behandla en klass `Vector` som ett genomgående exempel. Vi kommer att börja från en enkel klass och därefter överlagra olika operatorer för att göra klassen användbar och transparent. Klassen `Vector` representerar en matematisk vektor i rymdgeometri. Den innehåller x-, y- och z-koordinater samt metoder för att accessera dessa. En vektor definierar således en viss riktning och har en viss längd. Alla vektorer antas starta från origo, d.v.s. punkten (0, 0, 0).

Vi får en första definition av klassen `Vector` enligt följande:

```
#ifndef VECTOR_H
#define VECTOR_H
```

```
#include <iostream>
#include <math.h>

class Vector {
public:
  // konstruktorer
  Vector ();
  Vector (const Vector & V);
  Vector (const float X, const float Y, const float Z);

  // accessera individuella komponenter
  void setX (const float X) { m_X = X; };
  void setY (const float Y) { m_Y = Y; };
  void setZ (const float Z) { m_Z = Z; };
  float x () const { return m_X; };
  float y () const { return m_Y; };
  float z () const { return m_Z; };

  // get the length of the vector
  float length () const;

private:
  // x, y och z-komponenterna
  float m_X;
  float m_Y;
  float m_Z;
};

#endif  // VECTOR_H
```

Koden ovan finns i en headerfil `Vector1.h`. Implementationen av metoderna finns i filen `Vector1.cpp`. Den ser i grundutförande ut på följande sätt:

```
#include "Vector1.h"

Vector::Vector () {
  // nollställ alla medlemmar
  m_X = 0;
  m_Y = 0;
  m_Z = 0;
}

// copy-konstruktor
Vector::Vector (const Vector & V) {
  // kopiera data från 'V'
  m_X = V.m_X;
  m_Y = V.m_Y;
  m_Z = V.m_Z;
}

// skapa från separata värden
Vector::Vector (const float X, const float Y, const float Z) {
```

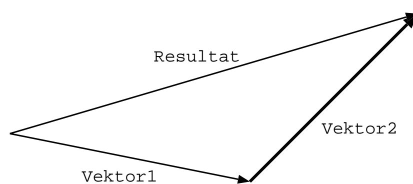
```
  // spara värden i medlemmar
  m_X = X;
  m_Y = Y;
  m_Z = Z;
}

// beräkna längden av vektorn
float Vector::length () const {
  // använd Pythagoras
  return sqrt ( m_X * m_X + m_Y * m_Y + m_Z * m_Z );
}
```

Senare då vi fyller på nya metoder skrivs inte båda filerna ut i sin helhet, utan endast prototypen på den nya (eller ändrade) metoden, samt implementationen om en dylik finns. Dessa skall då placeras i header- respektive implementationsfilen på korrekt plats. Korrekt plats i headerfilen är i klassens `public:`-avsnitt. Den kompletta klassen när den är färdig finns i Appendix C.

21.2. Överlagring av operatorn +

En fundamental funktion då man behandlar vektorer är möjligheten att addera vektorer. Vektorer adderas genom att placera dem efter varandra, varvid summan blir vektorn från den första vektorns startpunkt till den andra vektorns slutpunkt, enligt följande figur:

Figur 21-1. Vektoraddition



Addition av vektorerna `Vektor1` och `Vektor2` leder till resultatvektorn `Resultat`. Vill vi kunna göra denna operation med vår nuvarande definition av klassen `Vector` måste vi göra det manuellt med en speciell funktion eller metod som kan heta t.ex. `add()` eller något liknande. Vi kan istället överlagra operatorn `+`. Vi får då följande addition till klassen `Vector`:

```
// prototyp
Vector operator+ (const Vector & V) const;

// implementation
Vector Vector::operator+ (const Vector & V) const {
  Vector Result;

  // sätt den nya vektorns värden
  Result.setX ( m_X + V.m_X );
  Result.setY ( m_Y + V.m_Y );
  Result.setZ ( m_Z + V.m_Z );

  // returnera den nya vektorn
  return Result;
}
```

Prototypen skall alltså placeras headerfilen och implementationen i implemenationsfilen. Vi skall nu titta lite närmare på denna kryptiska metod. I C++ använder man nyckelordet `operator` för att berätta att man nu behandlar en operator av något slag. Man placerar den önskade operatorn efter nyckelordet `operator`, i vårt fall operatorn `+`. Detta blir alltså metodens namn. Som parameter tar metoden en annan `Vector`, som är den vektor vi vill addera till nuvarande vektor. Vi definierar `V` som en konstant referensparameter för att öka effektiviteten och eftersom vi inte vill ändra den på något sätt. Själva metoden är även `const`, eftersom vi inte ämna ändra nuvarande vektor heller. Däremot är returtypen `Vector`. Detta är resultatet av vår metod och alltså summan av de två vektorerna. Inne i metoden `operator+` definierar vi en ny `Vector` som vi tilldelar korrekta värden på de olika koordinaterna samt till sist returnerar.

21.2.1. Använda överlagrade operatorer

Vi har nu definierat en överlagrad operator `+`, men hur används denna? Som vi redan såg så heter metoden `operator+()`. Vi kan således använda klassen på följande sätt:

```
Vector V1 ( 1, 2, 3 );
```

```
Vector V2 ( 6, 5, 4 );

// addera vektorerna
Vector Result = V1.operator+ ( V2 );
```

Inte speciellt praktiskt, eller hur? Här kommer dock en del magi in i bilden. Man kan istället för att skriva `operator+` skriva endast + och även lämna bort parentesen. Ovanstående kod kunde alltså även skrivas:

```
Vector V1 ( 1, 2, 3 );
Vector V2 ( 6, 5, 4 );

// addera vektorerna
Vector Result = V1 + V2;
```

Nu börjar det lika någonting. Vi kan nu helt transparent addera vektorer till varandra. Vi kan även addera flera än två vektorer samtidigt:

```
#include <iostream>
#include "Vector2.h"

int main () {
  Vector V1 ( 1, 2, 3 );
  Vector V2 ( 6, 5, 4 );
  Vector V3 ( -1, -2, -3 );

  // addera vektorerna
  Vector Result = V1 + V2 + V3;

  cout « "x=" « Result.x () « ", "
       « "y=" « Result.y () « ", "
       « "z=" « Result.z () « endl;
}
```

Detta fungerar eftersom den första additionen av `V1` och `V2` returnerar en ny vektor som sedan i sin tur adderas med `V3`. Vi kunde använda parenteser för att framtvinga en viss evalueringsordning, men vektoraddition är kommutativ, så ordningen spelar ingen roll. Körning av programmet ovan ger oss följande (korrekta) utskrift:

```
% ./TestVector2_2
x=6, y=5, z=4
%
```

Alla överlagrade operatorer kan anropas utan `operator`*X*, där *X* är namnet på operatorn i fråga, men man kan göra det om man vill explicit visa att det är en överlagrad operator som anropas. Vi har här nått en viss transparens för vår klass `Vector`, nu återstår endel andra operatorer innan klassen är användbar i praktiken.

21.2.2. Överlagring av +=

Vi skall ännu i demonstrationssyfte överlagra operatorn `+=` för att skapa en metod som ändrar på objektet för vilket operatorn används. Vi kanske vill kunna skriva kod såsom t.ex.:

```
Vector V1 ( 1, 2, 3 );
Vector V2 ( 6, 5, 4 );

// addera vektor V2 till V1
V1 += V2;
```

Det görs med följande kod:

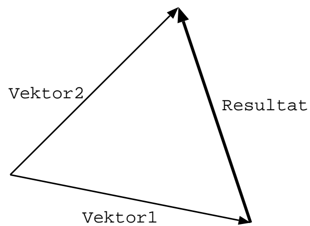
```
// prototyp
void operator+= (const Vector & V);

// implementation
void Vector::operator+= (const Vector & V) {
  m_X += V.m_X;
  m_Y += V.m_Y;
  m_Z += V.m_Z;
}
```

Metoden helt enkelt ändrar på medlemmarna för `V1` i exemplet ovan. Notera att metoden inte är definierad som `const` som de tidigare, eftersom den ändrar det anropande objektet.

21.3. Överlagring av operatorn -

Överlagring av den binära operatorn – är i princip en liknande operation som att överlagra + (se ovan). Enda skillnaden är hur de olika komponenterna kombineras. I detta fall subtraheras komponenterna från varandra.

Figur 21-2. Vektorsubtraktion



Vi kan definiera den överlagrade operatorn på följande sätt:

```
// prototyp
Vector operator- (const Vector & V) const;

// implementation
Vector Vector::operator- (const Vector & V) const {
  Vector Result;

  // sätt den nya vektorns värden
  Result.setX ( m_X - V.m_X );
  Result.setY ( m_Y - V.m_Y );
  Result.setZ ( m_Z - V.m_Z );

  // returnera den nya vektorn
  return Result;
}
```

Vi kan nu skriva kod som t.ex. följande:

```
Vector V1 ( 1, 2, 3 );
Vector V2 ( 6, 5, 4 );

// subtrahera vektorerna
Vector Result = V1 - V2;
```

Båda operatorerna + och - kan förstås kombineras ihop i diverse uttryck. Dessa evalueras enligt normal precedensordning, d.v.s. från vänster till höger om inte parenteser används för att indikera annan ordning.

Funderar vi lite närmare på operatorn - märker vi snabbt att den inte enbart fungerar som en binär operator för subtraktion, utan kan även användas *unärt*, d.v.s. men endast en parameter. I detta fall är det frågan om en negation av ett värde, eller i vårt fall, en vektor. Vi kan överlagra även den unära versionen av - i klassen `Vector`:

```
// prototyp
Vector operator- () const;

// implementation
Vector Vector::operator- () const {
  Vector Result;

  // sätt den nya vektorns värden
  Result.setX ( -m_X );
  Result.setY ( -m_Y );
  Result.setZ ( -m_Z );

  // returnera den nya vektorn
  return Result;
}
```

Metoden skapar en ny vektor som har ursprungsvektorns alla komponenter negerade. Vi får två metoder som heter `operator-`, men det är inget problem, eftersom de skiljer sig i vilka parametrar de accepterar, och kompilatorn vet således vilken metod som skall användas i olika situationer. Vi kan nu använda negationen i t.ex. följande kod:

```
Vector V1 ( 1, 2, 3 );

// negera vektorn
Vector Result = -V1;
```

21.4. Överlagring och friends

I normal vektoraritmetik kan man multiplicera vektorer med skalärer, d.v.s. vanliga tal för att ändra vektorns längd. Man multiplicerar då varje komponent med talet. Vi överlagra `*` på följande sätt:

```
// prototyp
Vector operator* (const float S) const;

// implementation
Vector Vector::operator* (const float S) const {
  Vector Result;

  // sätt den nya vektorns värden
  Result.setX ( m_X * S );
  Result.setY ( m_Y * S );
  Result.setZ ( m_Z * S );

  // returnera den nya vektorn
  return Result;
}
```

Vi kan nu skriva kod på t.ex. följande sätt:

```
#include <iostream>
#include "Vector5.h"

int main () {
  Vector V1 ( 1, 2, 3 );

  // multiplicera med skalär
  Vector Result = V1 * 3;

  cout « "x=" « Result.x () « ", "
       « "y=" « Result.y () « ", "
       « "z=" « Result.z () « endl;
}
```

Kör vi programmet ovan får vi följande resultat:

```
% ./TestVector5
x=3, y=6, z=9
%
```

Vi ser att varje komponent multiplicerats med tre, precis som vi avsåg. Men vad händer om vi istället skriver något i stil med följande:

```
Vector V1 ( 1, 2, 3 );

// multiplicera med skalär
Vector Result = 3.0 * V1;
```

Detta skulle innebära att datatypen `float` borde ha operatorn `*` överlagrad på ungefär följande sätt:

```
float.operator* (Vector);
```

Någon dylik funktion/metod finns inte. Vi märker alltså att det är den vänstra operanden som måste ha metoden överlagrad, och primitiva datatyper har inga överlagringar för icke-primitiva datatyper (såsom t.ex. `Vector` i vårt fall). Vi kan göra det enkelt för oss och säga att man måste ha skalärer till höger om en `Vector` då man utför multiplikation med skalär, men det är ointuitivt och leder till onödiga problem för programmeraren.

Lösningen är att använda en funktion som inte är medlem av klassen `Vector`.

21.4.1. Friend-funktioner

Eftersom det är omöjligt att definiera en metod för klassen `Vector` så att man kan multiplicera från vänster med en skalär skapar vi en funktion som är utanför klassen helt och hållet. Eftersom det är en vanlig funktion behöver man inget objekt för att anropa metoden, men istället tar den två parametrar, en `float` och en `Vector`. Vad vi gör är alltså att vi definierar om den generella operatorn `+` för argumenten `float` och `Vector`. Koden ser ut på följande sätt:

```
// prototyp
friend Vector operator* (const float S, const Vector & V);

// implementation
Vector operator* (const float S, const Vector & V) {
  // använd tidigare överlagrad operator '*'
  return V * S;
}
```

Notera att vi definierar *funktionen* som en *friend* till klassen `Vector` (för mera information om friends se Avsnitt 17.2). På det sättet kan funktionen accessera privata data i parametern `V` om så skulle behövas. Vi har dock gjort funktionen elegant och använder oss av den tidigare överlagrade operatorn `*` som tillhör klassen. För att kunna göra detta så reverserar vi ordningen på `S` och `V`, och får en multiplikation mellan en `Vector` och en `float`, vilket ju kan hanteras av den tidigare versionen.

Vi kan nu skriva t.ex. följande kod utan problem:

```
Vector V1 ( 1, 2, 3 );

// multiplicera med skalär
Vector Result = 3.0 * V1;
```

Ovanstående funktion blir om man använder sig av den oföskönade syntaxen följande:

```
Vector V1 ( 1, 2, 3 );

// multiplicera med skalär
Vector Result = operator* (3.0, V1);
```

Vi kunde om vi ville definiera alla överlagrade operatorer som funktioner på samma sätt som vi gjorde ovan, men det är inte speciellt vackert och bryter emot principen att kapsla ihop kod som har med en klass att göra med klassen själv. Viss funktionalitet måste dock implementeras via friend-funktioner.

21.5. Operatorerna == och =

En praktisk operator som ännu inte överlagrats är jämförelseoperatorn `==`. Vill vi med vår nuvarande definition av klassen `Vector` jämföra två vektorer måste vi manuellt jämföra varje medlem, vilket är fult och arbetsdrygt. Istället överlagrar vi operatorn `==` och implementerar den nödvändiga funktionaliteten:

```
// prototyp
bool operator== (const Vector & V) const;

// implementation
bool Vector::operator== (const Vector & V) const {
  // jämför V och denna vektor
  if ( m_X == V.m_X && m_Y == V.m_Y && m_Z == V.m_Z ) {
    // vektorerna är lika
    return true;
  }

  // de är olika
  return false;
}
```

Ett program som använder sig av `==` kan se ut på följande sätt:

```
#include <iostream>
#include "Vector8.h"

int main () {
  Vector V1 ( 1, 2, 3 );
  Vector V2 ( 6, 5, 4 );
  Vector V3 ( 1, 2, 3 );

  // jämför V1 och V2
  if ( V1 == V2 )
    cout « "V1 == V2" « endl;
  else
    cout « "V1 != V2" « endl;

  // jämför V1 och V3
  if ( V1 == V3 )
    cout « "V1 == V3" « endl;
  else
    cout « "V1 != V3" « endl;
}
```

Kör man programmet får man följande utskrift:

```
% ./TestVector8
V1 != V2
V1 == V3
%
```

Enda stora skillnaden mellan implementationen av denna överlagrade operator och de tidigare överlagrade är returtypen som är `bool`. Men vid närmare eftertanke är det ju den enda möjliga returntypen för en jämförelse som ju kan antingen vara sann eller falsk. Vi kan på samma gång överlagra även `!=` för att göra det lätt att jämföra vektorer med varandra:

```
// prototyp
bool operator!= (const Vector & V) const;

// implementation
bool Vector::operator!= (const Vector & V) const {
  // använd negerad jämförelse
  return ! ( *this == V );
}
```

Koden ovan kan behöva några förklarande ord. Vi använder oss av den tidigare överlagrade operatorn `==` så att vi inte behöver skriva en ny och lång `if`-sats, men negerar det resultat som vi får. Så om de två vektorerna är lika varandra så är det *inte*

olika varandra. Vi utför alltså först en jämförelse och sedan negeras resultatet. Här får vi nytta av att kunna referera till nuvarande objekt via pekaren `this`. Eftersom det inte finns någon överlagrad operator för att hantera pekare till vektorer måste vi först avreferera `this`, och först sedan utföra jämförelsen med `V`. Parentesen runt jämförelsen är nödvändig, eftersom `!` har högre precedens än `==`, och följdaktigen försöker kompilatorn först negera `*this` och sedan utföra jämförelsen. Vi vill dock göra det hela i motsatt ordning, därav parentesen.

21.5.1. Tilldelningsoperatorn =

En operator som vi egentligen inte behöver överlagra för vår enkla klass är operatorn `=`, d.v.s. normal tilldelning. Kompilatorn skapar alltid automatiskt en version av tilldelning ifall programmeraren inte gör det, och för klassen `Vector` skulle den versionen duga mycket väl. Det som den kompilatorgenererade versionen gör är att den skapar kod för att kopiera alla medlemmar från originalobjektet till det kopierade objektet. En dylik approach fungerar fint för vår klass, men kanske inte för en mer avancerad klass som innehåller pekare och/eller allokerat minne. Se t.ex. Avsnitt 14.3.5 för ett exempel på en klass som inte kan använda den normala tilldelningen utan att orsaka problem. Notera att klassen där även heter `Vector`, men där avses en container, inte en matematisk vektor.

Vi skall dock explicit överlagra operatorn `=` i demonstrationssyfte. För andra mer komplicerade klasser är definitionen i princip samma, men den egentliga koden inne i metoden skiljer sig förstås. Vi får följande metod:

```
// protoyp
Vector & operator= (const Vector & V);

// implementation
Vector & Vector::operator= (const Vector & V) {
  // tilldelas vi värdet av oss själva?
  if ( this == &V ) {
    // jep, gör inget i så fall
    return *this;
  }

  // utför tilldelning
  m_X = V.m_X;
  m_Y = V.m_Y;
  m_Z = V.m_Z;
```

```
  // returnera oss själva
  return *this;
}
```

Denna metod kräver även lite förklaring. För det första är returvärdet i denna metod en *referens* till en `Vector`. Alla andra metoder har returnerat en "vanlig" `Vector` och inte en referens. Skillnaden ligger i att de hade i så fall returnerar en referens till en lokal variabel, som ju skall förstöras direkt då metoden är avslutad. En referens till ett förstört objekt är inte giltig, därför returneras en kopia. Men vid överlagring av operatorn `=` returneras inte en referens till en loka variabel, utan till själva klassen för vilken metoden anropats, d.v.s. `this`. Vi avrefererar `this` för att returnera en `Vector` och inte en `Vector *`. Tilldelningen sker till aktuellt objekt, precis som med operatorn `+=`.

De första raderna i metoden kontrollerar att vi inte försöker tilldela objektet till sig själv, d.v.s. göra något i stil med:

```
Vector V1 ( 1, 2, 3 );

// tilldela till sig själv
V1 = V1;
```

Om så görs skall inget speciellt göras. Man kan kontrollera om adressen för `V` är samma som `this`. Om så är fallet är det samma objekt, eftersom de finns på samma adress i minnet.

Den skarpsynte funderar nu säkert varför måste vi returnera en referens till en `Vector` då vi överlagrar tilldelning, räcker det inte med att enbart kopiera data från ursprungsvektorn? Orsaken till detta är att man i C++ (och C) kan skriva multipla tilldelningar på samma rad, t.ex. är följande giltigt:

```
Vector V1, V2, V3;

// multipel tilldelning
V1 = V2 = V3 = Vector (1, 1, 1);
```

Här vill vi att `V1`, `V2` och `V3` skall alla få värdet av den sista vektorn. Tilldelningarna utförs från höger till vänster, så `V1` tilldelas sist. Normalt brukar man försöka undvika att använda multipel tilldelning, men eftersomd et är laglig C++ måste även en

överlagrad operator = understöda syntaxen. Om vi således inte returnerar en `Vector` i vår metod så "stoppas" sekvensen av tilldelningar.

Vi har nu definierat de flesta operatorer som är vettiga för en vektor. Man kan även lägga till annan funktionalitet såsom t.ex. rotationer, kryssprodukter o.s.v., men de lämnas som en övning till läsaren :-)

21.6. Överlagring av « och »

I våra exempelprogram hittills har vi använt medlemsvis access för att skriva ut innehållet i en `Vector`, d.v.s. vi har anropat metoderna `x()`, `y()` och `z()` för att komma åt koordinaterna i en vektor och skrivit ut dem. Vore det inte fint om klassen själv kunde skriva ut sig till skärmen eller annan stream (t.ex. en fil)? Även detta kan åstadkommas med hjälp av överlagring av operatorer. Vi måste dock även här använda en extern *funktion* som ges tillträde till `Vector` via en `friend`-deklaration. Koden kan se ut på följande sätt:

```
// friend-definition
friend ostream & operator« (ostream & Stream, const Vector & V);

// implementation
ostream & operator« (ostream & Stream, const Vector & V) {
  // skriv ut till den stream vi fått som parameter
  Stream « "x=" « V.x () « ", y=" « V.y () « ", z=" « V.z ();

  // returnera streamen
  return Stream;
}
```

Eftersom det är en funktion som inte tillhör klassen `Vector` måste även vektorn komma med som en parameter till funktionen. Se Kapitel 8 och Kapitel 19 för mera information om streams. Det kan verka konstigt att vi måste returnera `Stream` i funktionen. Det görs därför att man ju kan kombinera ihop långa utskrifter till en enda genom att placera multipla « och data emellan. Man måste returnera streamen så att nästa « skall kunna fortsätta skriva ut till en instans av `ostream`. I exemplet ovan skrivs alltså först `"x="` ut, sedan returneras `Stream` till nästa variabel/konstant som skall

skrivas ut, i det här fallet `V.x()`. Vad vi i praktiken använder är operatorn « som klassen `ostream` överlagrat för att möjliggöra enkel utskrift!

Vi kan nu enkelt skriva ut vektorer, t.ex. på följande sätt:

```
#include <iostream>
#include "Vector11.h"

int main () {
  Vector V1 ( 1, 2, 3 );
  Vector V2 ( 6, 5, 4 );

  // skriv ut V1 och V2
  cout « V1 « endl « V2 « endl;
}
```

Vi får då följande utskrift:

```
% ./TestVector11
x=1, y=2, z=3
x=6, y=5, z=4
%
```

21.6.1. Överlagrad input med »

På motsvarande sätt som man kan överlagra « för att möjliggöra enkel utskrift av vektorer kan man överlagra » för att möjliggöra inläsning av vektorer från tangentbord, filer eller annan stream. Här får vi dock ett problem med vår tidigare definition av operatorn « om vi vill kunna läsa in samma utskrivna data på nytt. Vi har i metoden koncentrerat oss på att göra utskriften lättläslig och informativ, men vi har samtidigt gjort den svårare att läsa in. Vi måste läsa i de i princip onödiga `"x="` o.s.v. Vi är dock endast ute efter att läsa endast tre tal, så vi ändrar koden för utskriftsoperatorn en aning:

```
ostream & operator« (ostream & Stream, const Vector & V) {
  // skriv ut till den stream vi fått som parameter
  Stream « V.x () « ' ' « V.y () « ' ' « V.z () « ' ';

  // returnera streamen
  return Stream;
}
```

Definitionen ändras inte, endast formateringen av den utskrivna texten. Vi skriver nu endast ut ett enda mellanslag som separerar de olika talen samt ett sista mellanslag efter z-koordinaten. Det sista mellanslaget hindrar att två vektorer skrivs ut fast i varandra, d.v.s. att den första vektorns z-koordinat skrivs ihop med den andras x-koordinat. Notera att vi inte behöver ändra definitionen på utskriftsmetoden ifall vi inte är intrrsserade av att använda enkel inläsning med ». Klasser har oftare överlagrat « än de överlagrat ». I dina egna klasser bör du överväga om en vacker eller lättläslig utskrift behövs. Detta gäller förstås endast då man skriver till filer, eftersom man inte kan läsa in någonting från *skärmen*.

Vi kan nu definiera operator » som läser in data som vår modifierade utskriftsoperator ovan skrivit ut:

```
// friend-definition
friend istream & operator» (istream & Stream, Vector & V);

// implementation
istream & operator» (istream & Stream, Vector & V) {
  // skriv ut till den stream vi fått som parameter
  Stream » V.m_X »  V.m_Y » V.m_Z;

  // returnera streamen
  return Stream;
}
```

Denna överlagrade operator är implementerad enligt samma principer som överlagrade «, men vi läser istället in i en vektor `V`. Vi märker att `V` inte är deklarerad som `const`, eftersom vi ämnar ändra dess värde. Då vi läser in behöver vi inte bry oss om att "läsa bort" de tomrum som vi skrev ut mellan vektorns koordinater, eftersom det görs automatiskt av `istream`. Vi kan således göra t.ex. följande program som skriver ut två vektorer till en fil och läser in dem igen, men i omvänd ordning (en form av *swap* utan temporär variabel...):

```
#include <stream>
#include "Vector12.h"

int main () {
  Vector V1 ( 1, 2, 3 );
  Vector V2 ( 6, 5, 4 );

  // öppna en fil
  ofstream Ut ( "VektorData.txt" );
```

```
  // skriv ut V1 och V2
  Ut « V1 « endl « V2;

  // stäng utfilen
  Ut.close ();

  // öppna filen på nytt för läsning
  ifstream In ( "VektorData.txt" );

  // läs in till vektorerna, men i omvänd ordning
  In » V2 » V1;

  // skriv ut på skärmen
  cout « "V1: " « V1 « endl « "V2: " « V2 « endl;
}
```

Kör vi programmet får vi följande utskrift på skärmen:

```
% ./TestVector12
V1: 6 5 4
V2: 1 2 3
%
```

Vi märker att `V1` och `V2` bytt värden, eftersom vi läste in dem i omvänd ordning mot när vi skriv ut dem.

21.7. Överlagring av typkonverteringar

Skall skrivas!

21.8. Diskussion

Vi har nu gått igenom grunderna i hur operatorer kan överlagras och sett på fördelarna man kan uppnå. Så länge som man använder överlagring av operatorer på ett vettigt sätt blir klasser mycket lättare att använda, och följdaktligen program lättare att läsa. Man har mycket att vinna genom korrekt hantering av överlagring. I fel händer är det dock

frågan om ett formidabelt vapen som kan åstadkomma mycket oreda och problem i program. Speciellt om man försöker ändra på de ursprungliga operatorernas betydelse.

Det finns många olika operatorer som vi inte ens tittat på, men som för många klasser är vettiga, t.ex. operatorerna `++` och `--`, men de är relativt lätta att implementera med hjälp av exemplen i detta kapitel.

Ett bra exempel på klasser där överlagring av operatorer använts i mycket stor skala är de olika klasserna i *Standard Template Library* (se Kapitel 24). Där överlagras en stor mängd olika operatorer, såsom `=`, `[]`, `++`, `--` samt många andra. Utan överlagrade operatorer skulle program som använder sig av STL vara ganska svårlästa, men nu är komplexiteten delvis gömd bakom ett enhetligt gränssnitt som uppnåtts genom noggrann överlagring av operatorer.

Den kompletta klassen för `Vector` finns i Appendix C.

Kapitel 22. Serialisering

I detta kapitel redogörs för hur man kan spara objekts status på t.ex. disk via *serialisering*.

Kapitel 23. Typparametrisering

I detta avsnitt redogörs för grunderna i hantering av generiska klasser, s.k. *template-klasser*.

23.1. Iden med templates

De flesta program behöver spara data i någon form under sin körtid. Vanligen används någon form av datastruktur för detta data, t.ex. en vektor, lista, kö, stack, hashtabell el.dyl. Det är relativt vanligt att dessa datastrukturer inkapslas i en klass så att de skall vara lättare att använda och utvidga eller specialisera via ärvning. Vi kan ta som exempel den *stack* som fungerade som exempel i Kapitel 12. Vi moderniserar den så att vi får en en *Stack*-klass:

```
class Stack {
public:
  // exceptions
  enum Exception { Underflow };

  // konstruktor
  Stack () : Top(0) { };

  // metoder för stackmanipulation
  void push (const char Data);
  char pop ();

private:

  // ett element i stacken
  struct Element {
// data för detta element
char Data;
// nästa element under detta i stacken
Element * Previous;
  };

  // stackens topp
  Element * Top;
};

// push:a ett element på stacken
void Stack::push (const char Data) {
  // skapa nytt element
```

```
  Element * New = new Element;
  New->Data = Data;
  New->Previous = Top;

  // nytt toppelement
  Top = New;
}

// pop:a ett element från stacken
char Stack:: pop () {
  // är stacken tom?
  if ( Top == 0 ) {
// stacken tom
throw Underflow;
  }

  // spara data som är lagret i elementet
  char Data = Top->Data;

  // spara pekare till det element som blir nytt topp-element
  Element * Tmp = Top;
  Top = Top->Previous;

  // frigör minne och återvänd
  delete Tmp;
  return Data;
}
```

Denna stack är totalt dynamisk och enkel att använda. De enda metoderna som finns är metoder för att lägga till ett element och ta bort det översta elementet. Den använder sig av en exception (se Kapitel 20) för att rapportera ett fel om man försöker poppa ett element ur en tom stack. Denna stack fungerar dock endast med data av typen `char`. Det är inte svårt att ändra om den att fungera med olika datatyper, det enda som behöver ändras är själva datatypen, all övrig kod hålls intakt. Vi kunde skapa olika versioner för olika datatyper, t.ex. `CharStack`, `IntStack` och `StringStack`. Låter det som slöseri med tid och minne, eftersom de olika klasserna är i princip identiska? Blir det inte problem med att hålla alla klasser uppdaterade om det blir ändringar?

Det hela går att sköta enklare och elegantare med *templates* (typparametrisering). Med typparametrisering menas att man gör den datatyp som en klass opererar på som en parameter. Vi kan alltså skapa en `Stack` och ge den typ vi vill ha som en parameter. Kompilatorn genererar sedan åt oss en stack som använder sig av just den datatyp vi vill använda. Vi kan således enkelt använda stackar för olika datatyper i samma program, men använda endast en version av koden.

Templates används oftast för att skapa *containers*, d.v.s. klasser som är menade att innehålla objekt av någon typ. I och med att containers vanligen är ganska generiska och olika typer inte kräver speciellt mycket olika kod (om alls) används templates i hög grad. Man kan även använda templates för att kapsla in t.ex. generiska algoritmer. Templates är det närmaste C++ kommer då det gäller att skriva kod som är typoberoende, d.v.s. där samma kod kan användas för olika datatyper. Vi kommer senare att se på en hel del olika fördefinierade templates i Kapitel 24.

23.2. Ett konkret exempel

Vi skall nu visa hur man kan typparametrisera vår klass `Stack` så att den accepterar vilken datatyp som helst:

```
template<class T>
class Stack {
public:
  // exceptions
  enum Exception { Underflow };

  // konstruktor
  Stack () : Top(0) { };
  ~Stack ();

  // metoder för stackmanipulation
  void push (const T & Data);
  T pop ();

private:
  // ett element i stacken
  struct Element {
    // data för detta element
    T Data;
    // nästa element under detta i stacken
    Element * Previous;
  };

  // stackens topp
  Element * Top;
};

// destruktor
template<class T>
Stack<T>:~Stack () {
  // radera alla element
```

```
  try {
while ( pop () );
  }
  catch ( Exception E ) {
// stacken tom
  }
}

// push:a ett element på stacken
template<class T>
void Stack<T>::push (const T & Data) {
  // skapa nytt element
  Element * New = new Element;
  New->Data = Data;
  New->Previous = Top;

  // nytt toppelement
  Top = New;
}

// pop:a ett element från stacken
template<class T>
T Stack<T>::pop () {
  // är stacken tom?
  if ( Top == 0 ) {
    // stacken tom
    throw Underflow;
  }

  // spara data som är lagret i elementet
  T Data = Top->Data;

  // spara pekare till det element som blir nytt topp-element
  Element * Tmp = Top;
  Top = Top->Previous;

  // frigör minne och återvänd
  delete Tmp;
  return Data;
}
```

Man definierar en templateklass genom att placera `template<class T>` framför klassdeklarationen. Parametern `T` kan heta vad som helst, men ofta används `T` eller `Type`. Här är `T` en parameter som representerar den datatyp som stacken skall använda. Det är en helt *generisk typ*, d.v.s. den kan vara vilken datatyp som helst. Alla förekomster av `char` har ersatts med vår nya typ `T`. Samma `template<class T>` måste även placeras före metoddefinitionerna, samt även ett `<T>` mellan klassens namn och `::`. Det hela ser tyvärr ganska fult ut. Man kan om man vill skriva `template<class T>` på samma rad som själva metod- eller klasdefintionen, men

detta sätt är det som förekommer mest. Datatypen `T` används som parameter, returvärde, lokala värden och som del i den `struct` som definierats. Man kan tolka `T` som en helt vanlig datatyp.

23.2.1. Instantiering av templateklasser

Nu har vi alltså definierat en första templateklass, nu är det dags att använda den också. Som man kunde gissa är även där syntaxen lite speciell. För att instantiera en `Stack` som fungerar på datatypen `int` kan vi använda följande sats:

```
Stack<int> IntStack;
```

Denna kan sedan användas efter deklarationen precis som en vanlig stack utan extra "påhäng". Om man vill skicka stacken som en parameter till en funktion el.dyl. måste man använda det fulla namnet `Stack<int>`. Vi kan även instantiera en strängstack och en floatstack:

```
Stack<string> S1;
Stack<float> Values;
```

Vikan göra ett enkelt program som använder sig av en stack av `int` enligt följande (det är en adaption av programmet i Kapitel 12):

```
int main () {
  Stack<int> S;
  char Choice = ’ ’;
  int Data;

  // iterera tills användaren vill avsluta
  while ( Choice != ’3’ ) {
    cout « "Välj: " « endl « "1 - push" « endl « "2 - pop" « endl;
    cout « "3 - sluta " « endl « "-> ";

    // läs in ett menyval
    cin » Choice;

    // vad vill användaren göra
    switch ( Choice ) {
    case ’1’ :
       cout « "Elementets värde: ";
       cin » Data;
       S.push ( Data );
```

```
        break;

    case '2' :
      // försök poppa ett element
      try {
        Data = S.pop ();
        cout « "Poppade: " « Data « endl;
      }
      catch ( Stack<int>::Exception E ) {
        // stacken top
        cout « "Stacken tom!" « endl;
      }
      break;
    }
  }
}
```

Notera att stacken är definierad som `Stack<int> S;` och att vi således måste även använda samma typdefinition då vi fångar en excaption, alltså `Stack<int>::Exception E`.

Vi kan själv klart använda oss av olika instatieringar av vår `Stack`-klass i samma program, även samma funktion eller metod.

23.3. Multipla parametrar

Exemplet ovan använder sig av endast en typparameter, men det finns inget som hindrar att man använder två eller flera. De olika typerna är totalt oberoende av varandra. Vi kan se på en enkel klass som lagrar *par* av data, t.ex. en person och dennas socialskyddsnummer.

```
template<class T1, class T2>
class Pair {
public:
  // konstruktor
  Pair (const T1 & V1, const T2 & V2) : m_First(V1), m_Second(V2) { };

  // accessera de olika värdena
  T1 first () { return m_First; }
  T2 second () { return m_Second; };

private:
  // våra två element
```

```
  T1 m_First;
  T2 m_Second;
};
```

Denna klass är så enkel att alla metoder är definierade direkt i dess `class`-deklaration. Vi har två typer, `T1` och `T2` som definierar de datatyper som vi kan göra par av. Det är ingen skillnad vilka datatyper vi använder oss av. Vi separerar de två typerna med hjälp av ett kommatecken i `template<class T1, class T2>` -definitionen. Ett enkelt exempel som instantierar en klass som använder sig av `int`-`string`-par:

```
int main () {
  Pair<int,string> * Persons[2];
  int Id;
  string Name;

  // iterera och mata in data om personerna
  for ( int Index = 0; Index < 2; Index++ ) {
    cout « "Ge id: ";
    cin » Id;
    cout « "Ge namn: ";
    cin » Name;
    Persons[Index] = new Pair<int,string> ( Id, Name );
  }

  // skriv ut våra personer
  for ( int Index = 0; Index < 2; Index++ ) {
    cout « "Person " « Index;
    cout « ", id: " « Persons[Index]->first ()
         « ", name: " « Persons[Index]->second () « endl;

    // frigör minne
    delete Persons[Index];
  }
}
```

Programmet är ganska enkelt att förstå. Vi deklarerar här en vektor som innehåller `Pair<int, string>`.

Det är även fullt möjligt att använda templates som har andra templates som typer. Vi kan t.ex. definiera en stack av stackar av `double` med följande anrop:

```
Stack< Stack<double> > WeirdStack;
```

Varje element i den "yttre" stacken är nu en annan stack som hanterar parametrara av typen `double`. Notera att man måste sätta minst ett mellanrum mellan de två > i definitionen för att särskilja definitionen från inputoperatorn ».

23.3.1. Definiera lättare namn

Det kan i ibland bli ganska svåra och oöverksådligt långa namn på datatyper. Det är enkelt att tappa bort vilket som är namnet på datatyperna och vilket är namnet på den egentliga variabeln. Vi kan ta som exempel en prioritetskö som använder sig av datatypen `Element *`. Den normala definitionen när vi vill deklarera en variabel vore då t.ex.:

```
PriorityQueue<Element *> Queue1;
```

Vi kan med hjälp av en `typedef` göra namnet lättare att använda så vi slipper de fula < och > som följer med templateinstantieringar:

```
// definiera en lättare typ
typedef PriorityQueue<Element *> ElementQ;

// använd denna nya typ
ElementQ Queue1;
```

En hel del enklare eller hur? Detta är en av de få situationer där man normalt använder sig av `typedef`.

Kapitel 24. Standard Template Library

Detta kapitel behandlar en annan ny del av standarden för C++, nämligen *Standard Template Library*, som namnet säger är ett bibliotek med standardiserade template-klasser.

24.1. Vad är STL

STl är en förkortning av *Standard Template Library*, men är lika känt under nanmnet *STL*, så förkortningen används här.

Som redan nämndes i kapitlet om typparametrisering (se Kapitel 23) är templates mycket praktiska då det gäller att skapa generiska containers och klasser som kan operera på många olika datatyper. De flesta större program behöver någon form av lagring för diverse former av data och templatiserade containers (classer skapade för att "innehålla" andra klasser) är mycket praktiska. För de flesta större program skapades alltså egna container-klasser, som kanske återanvändes i andra projekt, eller så skapades nya för nästa projekt. Slutresultatet var att man "uppfann hjulet gång på gång", och alltid skapade nytt som aldrig fick en chans att bli totalt buggfritt och använt i stora kretsar. Kommiten som standardiserar C++ märkte detta och tog därför med ett bibliotek med olika template-klasser i standarden för C++. Dessa klasser är menade att skall fungera som bas för de mesta av container-behov ett program kan ha. Så föddes STL, som numera finns med i varenda modern C++-kompilatorn.

Tanken med STL är att om det finns ett standardiserat biliotek så undviker man att uppfinna hjulet varje gång, och kan istället koncentrera sig på att programmera själva logiken i ett program. STL har även fått en chans att ordentligt stabiliseras och bli både buggfritt och väl optimerat. Man har velat göra klasserna i STL snabba och minnessnåla så att program inte "bestraffas" för att de använder sig av klasser ur STL.

Detta material kan endast skrapa på ytan på den funktionalitet som finns i STL, och biblioteket skulle egentligen kräva ett helt eget kompendium för att ge en grundläggande översikt. Några metoder för vissa klasser visas, men läsaren hänvisas till de extensiva referenshemsidor som nämns i kapitlet *Referenser*.

24.1.1. Grundblock i STL

STL är ett stort bibliotek men många klasser och ännu flera användningsområden. De klasser som finns i STL kan grovt indelas i några stora huvudgrupper:

- *container-klasser* som används för att hålla ordning på objekt. Här finns bl.a. listor, köer stackar och vektorer.
- *generiska algoritmer* som är algoritmer som opererar på containers. Dessa algoritmer fungerar på samma sätt oberoende av vilken container de används på. Exempelvis sortering är en generisk algoritm.
- *iteratorer* som används för att iterera igenom de objekt som finns i en container. Alla containers ha egna iteratorer, men deras gränssnitt ser lika ut, så användaren kan således enkelt iterera genom objekten i vilken container som helst.
- *funktionsobjekt* som fungerar som funktioner som appliceras på en container. Dessa kan utföra någon viss operation på varje element, t.ex. beräkna ett medeltal eller summor.
- *adaptorer* som adapterar gränssnittet som någon klass har för att passa andra behov, t.ex. baklängesiteratorer.

Förutom dessa finns ännu olika former av allokatorer som används för att allokera minne för de olika klassernas interna bruk. Dessa behöver man sällan befatta sig med. De flesta olika klasserna av dessa kategorier kan ganska fritt "pluggas ihop" med varandra utan skillnad. Detta är styrkan bakom generisk programmering och STL. I de följande avsnitten behandlas de olika grundblocken lite mera ingående.

24.1.2. Effektivitet

Alla operationer som sker med containers och funktioner i STL har optimerats för maximal effektivitet. I mera ingående dokumentation redogörs exakt för den tid det tar att accessera, radera och sätta in element i en container, samt tiden det tar att utföra olika funktioner. Dessa ges med den normala *O-notationen* (se t.ex. kursen i datastrukturer). För de flesta operatione finns även för- och eftervillkor definierade,

d.v.s. vad som finns före en operation utförs, och vilket "läge" containern befinner sig i efter operationen (se t.ex. kursen i programmeringsmetodik). Denna effektivitetsinformation kommer inte att tas upp i detta material, utan det är information för en kurs i datastrukturer eller STL-programmering.

24.2. Containers

Den kanske viktigaste delen i STL är nog de olika *containers* som finns implementerade. Dessa containers fyller de flesta programs behov av klasser för att spara data i. En mängd olika containers med olika användningsområden finns. Dessa kan indelas i två grupper beroende på hur data i dem kan accesseras.

24.2.1. Sekvenscontainers

Dessa containers har fått sitt namn av att data är lagrat i strikt sekventiell ordning. Man kan accessera all data i en slumpmässig ordning. De olika containers som finns är:

- `vector` som fungerar som en normal vektor, men den är mera dynamisk och kan ändra storlek om det behövs. Normala vektorer har en fast storlek. Access av data är snabbt, d.v.s. i konstant tid. Använd dessa istället för normala vektorer om storleken kan komma att ändra under exekveringen.
- `deque` är en implementation av en normal kö, där data kan effektivt sättas in och raderas i början och slutet. Kan användas då man behöver t.ex. en FIFO-kö för att organisera data för processering.
- `list` är en vanlig lista av element. Access till data i mitten av listan sker i linjär tid. Kan användas där man behöver en lista med element och inte behöver göra alltför frekventa accesser till data i mitten av listan. Insättningar och raderingar i mitten av en lista är dock effektiva. En lista kan inte accesseras i slumpmässig ordning.

24.2.2. Sorterade associativa containers

Associativa containers skiljer sig från normala containers i och med att man i här använder sig av en `nyckel` för att accessera data. Ett naturligt exempel kan vara att lagra personer i ett register på bas av t.ex. en medlemsnummer aller deras socialskyddssignum. All data sorteras på bas av nyckeln. Om man använder en nyckel som inte kan (eller bör) direkt jämföras med == kan man till konstruktorn ge med en egen jämförelsefunktion. De associativa containers som kan användas är:

- `set` som definierar en enkel klass för hantering av en mängd av data. Denna är sorterad på bas av en nyckel.
- `map` fungerar på samma sätt som `set`, men innehåller end del extra metoder som gör den lättare att använda och den klass som normalt används.
- `multiset` är ett `set` som tillåter multipla element att ha samma nyckel.
- `multimap` är en `map` som tillåter multipla element att ha samma nyckel.

24.2.3. Instättning och access av data

Att sätta in data i sekvenskontainers är enkelt, men varierar lite beroende på vilken container som används. Som exempel visas hur data läses in från tangentbordet kan placeras i en `vector` ända tills strängen `end` läses in:

```
#include <vector>
#include <string>

int main (int argc, char * argv[]) {
  vector<string> Data;
  string Value;

  cout « "Skriv text och avsluta med ’end’:" « endl;

  // läs data från tangentbordet
  while ( cin » Value ) {
    // är vi klara med insättngen?
    if ( Value == "end" ) {
      // jep, hoppa ut
      break;
    }
```

```
    // addera till vektorn
    Data.push_back ( Value );
  }

  // iterera över alla värden
  for ( unsigned int Index = 0; Index < Data.size (); Index++ ) {
    cout « Index « ": " « Data[Index] « endl;
  }
}
```

Programmet visar hur man med hjälp av metoden `push_back` placerar data i slutet på vektorn.

Vi kunde byta ut `vector` i exemplet ovan till `deque` och få exakt samma resultat, eftersom båda har samma metoder definierade, och båda kan accesseras slumpmässigt via `operator[]`.

Exempel på användning av en associativ container (`map`) finns nedan. Vi skapar här ett program som räknar förekomsten av ord i en fil, och skriver sedan ut en frekvenstabell. som nyckel används själva ordet, medan dte egentliga datat för varje element är en `int` som anger antalet förekomster. Orden är automatiskt sorterade. I en `map` kan man sätta in element med `operator[]`. Om det redan finns ett element med samma nyckel skrivs det över. Om man vill tillåta flera element med samma nyckel kan man använda `multimap` och `multiset`.

```
#include <map>
#include <string>
#include <fstream>
#include <stdlib.h>

int main (int argc, char * argv[]) {
  // definiera en 'map' av string->int
  map<string, int> Text;
  string Value;

  // har vi parametrar?
  if ( argc != 2 ) {
    cout « "Fel antal parametrar!" « endl;
    cout « "Användning: " « argv[0] « " filnamn" « endl;
    exit ( EXIT_FAILURE );
  }

  // öppna en fil
  ifstream Data ( argv[1] );
  if ( ! Data ) {
    // kunde inte öppna filen
```

```
    cout « "Kunde inte öppna filen: " « argv[1] « endl;
    exit ( EXIT_FAILURE );
  }

  // iterera över all ord i filen
  while ( Data » Value ) {
    // lagra i vår map
    Text[Value]++;
  }


  // iterera över våra ord och skriv ut deras frekvenser
  map<string, int>::iterator It = Text.begin ();
  for ( ; It != Text.end (); It++ ) {
    // en 'map' innehåller 'pair':s som innehåller vår data
    pair<string, int> P = *It;
    cout « P.second « "\t" « P.first « endl;
  }

  cout « endl « "Totalt " « Text.size () « " unika ord." « endl;

  // close stream
  Data.close ();
}
```

Exemplet ovan visar även filhantering med hjälp av klassen `ifstream`.

De flesta klasser har metoder för att placera data först och sist i containern med hjälp av metoderna `push_front` och `push_back`. Vissa kan även använda `operator[]`. Man kunde även använda metoden `insert()` som använder sig av en *iterator* (se Avsnitt 24.3) för att berätta var i containern det nya elementet skall placeras.

24.2.4. Radering av data

De flesta containers har metoder för att radera data. Man kan i de flesta fallen använda både `pop_front()` och `pop_back()` för att radera ett element. Denna metod kommer dock inte att frigöra minne som elementet eventuellt allokerat, utan man måste manuellt förstöra elementet med t.ex. `delete`. Exemplet nedan visar hur man kan göra:

```
#include <map>
#include <string>
#include <iostream>

int main (int argc, char * argv[]) {
  map<int, string *> PhoneBook;
```

```
  string Value;

  // sätt in element i vår telefonbok
  PhoneBook[5551234] = new string ("Foo Bar");
  PhoneBook[5555678] = new string ("Joe Q. Public");
  PhoneBook[5553456] = new string ("John Smith");
  PhoneBook[5559876] = new string ("Hugo Hacker");

  cout « "Vi har " « PhoneBook.size () « " personer." « endl;

  // vi vill radera ett element från vår telefonbok
  string * DeleteMe = PhoneBook[5553456];
  delete DeleteMe;

  // utför raderingen
  PhoneBook.erase ( PhoneBook.find (5553456));

  cout « "Vi har nu " « PhoneBook.size () « " personer." « endl;
}
```

Detta program opererar på data som är `string *` och som allokeras då det sätts in. Om vi när vi raderar ett element från listan inte även frpgör minnet kommer vårt program att läcka minne. Vi accesserar elementet med `operator[]` och raderar det, och sedan raderar vi själva elementet i containern med `erase()`. Erase tar som parameter en `iterator` som visar vilket element som skall raderas, och det elementet hittar vi med t.ex. `find()`. Vi kunde alternativt skrivit:

```
// vi vill radera ett element från vår telefonbok
map<int, string *>::iterator It = PhoneBook.find ( 5553456 );
delete (*It).second;

// utför raderingen
PhoneBook.erase ( It );
```

Denna version är effektivare, eftersom vi nu utför endast en sökning. Som vi redan såg i ett tidigare exempel användes en klass `pair` för att accessera det data som finns i ett element. I en associativ container är varje element egentligen en instans av `pair`, som är templatiserad att använda samma argument som containern. En instans av pair innehåller två objekt, det första är nyckeln (accessera via medlemmen `first`) och det andra själva data (accessera via `second`). I vårt fall ovan kan man avreferera `It` för att få den `pair` som innehåller uppgifterna. Vår `map` ser ut som bilden nedan illustrerar:

Figur 24-1. Uppbyggnaden av en `map`

map<int,string *>
Phonebook
pair<int,string *>
first second "Foo Bar"
5551234
first second "Joe Q. Public"
5555678
first second "John Smith"
5553456

Det kan verka svårt till en början att förstå alla avrefereringar och returnvärden rätt, men efter en tids användande tycker man vanligen att det sätt på vilket STL fungerar är naturligt.

24.2.5. Övrig funktionalitet hos containers

Det finns även annan funktionalitet hos containers. Den mest använda är troligtvis att få ut antalet element som en container innehåller. Detta kan göras såsom exemplen ovan visar med hjälp av metoden `size()`. Man kan i de flesta fall jämföra om två containers innehåller samma element genom den överlagrade operatorn `==`. Det är även enkelt att kopiera containers via en konstruktor som tar som parameter en existerande container. Detta fungerar dock endast då både originalet och den blivande kopian är av samma typ.

En viktig och ofta använd metod är `find()` som försöker hitta ett element i en associativ container på bas av en nyckel. Dett är den normala metoden för att söka element.

24.3. Iteratorer

Iteratorer är den nästvikigaste gruppen av klasser i STL. Via iteratorer ges programmeraren åtkomst till alla element i en container. Man kan se på en iterator som en "pekare" in i en container som pekare på ett element. Via denna pekare kan man sedan accessera elementet i just denna position. Multipla iteratorer kan användas på samma container, de kan t.o.m. peka på samma element. Iteratorer är enkla klasser men de innehåller funktionalitet för att gå till nästa eller föregående element i containern, för att på så vis ge möjligheter att enkelt iterera igenom alla element. Alla containers stöder iteratorer.

Det finns några olika typer av iteratorer som fungerar på olika sätt. Alla iteratorer stöds inte av alla containers beroende på det sätt containern är implementerad. De olika typerna är:

- *framåt-iteratorer* kan endast iterera åt ett håll. Från det ställe där de startar kan de endast gå mot den andra ändan av containern. T.ex. en lista itereras enklast igenom med en dylik iterator. Man kan gå antingen framåt eller bakåt, men inte åt båda hållen.
- *framåt och bakåtgående iteratorer* kan iterera framåt och bakåt i den container de opererar på. Alla containers kan inte använde denna typ av iteratorer.
- *random access-iteratorer* som kan accessera en container på vilket sätt som helst, samt hoppa från element till ett annat utan att de måste vara åtföljande element.

24.3.1. Använda iteratorer

Ett exempel på hur man kan iterera igenom en container ges nedan:

```
deque<float> Scores;
...
deque<float>::iterator Start = Scores.begin ();
deque<float>::iterator End = Scores.end ();
for ( ; Start != End; Start++ ) {
  // skriv ut värdet
  cout « *Start;
}
```

Varje container har metoderna `start()` och `end()` för att returnera iteratorer som pekar på det första respektive förbi det sista elementet. Men förbi det sista elementet avses en iterator som itererat ända till slutet och sedan "tar ett steg" till. Denna iterator innehåller ett illegalt värde och kan användas för att kontrollera när man itererat genom hela containern. Många metoder returnerar `end()` för att indikera att någonting inte kunde utföras eller hittas. Metoden `find()` returnerar `end()` om det sökta elementet inte kunde hittas.

Många av de olika metoderna för containers använder sig av iteratorer för att markera vilket element som skall accesseras. T.ex. metoderna `erase()` och `insert()` använder sig av iteratorer för att markera det element som skall raderas respektive den plats där ett nytt element skall placeras in. Metoden `erase()` finns vanligen i en version som tar två iteratorer som parametrar. Dessa markerar då det första och det sista elementet i en mängd element som skall raderas.

24.4. Generiska algoritmer

Den tredje stora gruppen i STL är de olika *generiska algoritmerna*. Dessa algoritmer operera på containers och utför någon viss funktionalitet. Det som gör de speciella att nästan alla algoritmer accepterar containers av olika typ som parametrar. Man kan således använda en funktion `find()` till att söka i t.ex. en `vector`, en `map` eller en `list`. Det är utanför detta material somfång att närmare redogöra för vilka olika funktioner som finns och hur de används, utan läsaren refereras till den externa litteraturen för mera information.

Kort kan dock nämnas vilka olika huvudfunktioner som existerar bland de generiska funktionerna. En huvudgrupp av funktioner är de som är avsedda för sökning av element. Funktioner i denna grupp är bl.a. `find()`, `search()` och `binary_search()`. En annan huvudgrupp är funktioner för att utföra mängdoperationer på `sets`. Man kan utföra t.ex. `includes`, `set_union`, `set_difference` och `set_intersection`.

Olika funktioner för att sortera (`sort()`) och transformera en container på olika sätt

finns också tillgängliga. Bl.a. kan man reversera (`reverse()`), partitionera, ersätta element (`replace()`), fylla en container med element (`fill()`) o.s.v. Det finns en mängd andra mer eller mindre användbara funktioner.

24.5. Adaptorer

En *adaptor* är ett gränssnitt som placeras "ovanpå" en container för att emulera någon annan typ av container som inte direkt finns implementerad som en egen klass. Det finns tre olika nya containers som kan användas via adaptorer, nämligen `stack`, `queue` och `priority_queue`. Olika klasser kan användas som "bas" för adaptorerna, men vissa är mer lämpliga än andra och alla klasser kan inte användas för allting. Normalt används endast sekvenscontainers. För att t.ex. skapa en stack av `float` som använder en lista som bas kan vi använda följande anrop:

```
stack<list<float> > Scores;
```

Notera igen att man måste ha ett mellanrum mellan de båda > för att undvika kollision med I/O-operatorn ». Notera även att den stack vi skapade i Kapitel 23 kunde enkelare gjorts med en `stack` ut STL!

En `queue` är en normal mö men där ingen iteration över elementen tillåts. Man kan endast accessera det första och det sista elementet. Insättningar och raderingar sker likaså i början eller i slutet av en `queue`. En `queue` skapas på samma sätt som en `stack`.

En `priority_queue` är även en speciell form av kö ur vilken man kan hämta element i storleksordning. Prioritetsköer används i flera speciella sammanhang, t.ex. för skedulering av processer.

24.6. Diskussion

Som redan flera gånger tidigare nämnts är det mycket svårt att ge en vettig översikt av

STL i detta kompendium. STL är kanske den största enskilda komponenten i C++ och är därmed mycket viktig. För en nybörjare kan STL verka svårt och ha en underlig syntax i vissa fall. Det stämmer att STL är en aning svårt att komma in i, men när man väl kommit över en första tröskel och kan använda STL utan att behöva kolla upp syntaxen på varenda en metod blir det en mycket trevligare att görs STL:iserade program.

I och med att det nu finns ett bibliotek som har en mängd standardiserad templatiserade containers är det dumt att skapa nya i egna projekt. Det finns förstås behov där STL helt enkelt inte är praktiskt att använda, men de torde vara ganska få. Programmeraren kan i och med STL koncentrera sig på *vad* som skall göras med data, inte *hur* det skall lagras i minnet. På så vis får program en kortare utvecklingstid och i de flesta fall mindre buggar.

Kapitel 25. Namespaces

Detta kapitel behandlar hur man kan gruppera kod i logiska "namnrymnder" för att undvika att klasser och funktioner med samma namn skall orsaka problem.

25.1. Vad är ett *namespace*

Namespaces används i C++ för att gruppera kod i logiska helheter. De fungerar på en högre nivå än klasser och funktioner, och används för att just gruppera klasser, funktioner och andra definitioner i en gemensam namnrymd. Genom att gruppera kod i namespaces kan man undvika problem som kan framkomma i stora projekt, nämligen att t.ex. samma klassnamn syftar till två eller flera helt olika klasser. Man får då problem med kompileringen av projektet. Med hjälp av namespaces kan man gruppera olika delar av ett stort projekt i olika namespaces och därmed undvika kollisioner.

25.1.1. Exempel på namnkollision

Vi kan titta på ett exempel på ett program där en namnkollision förekommer. Vi antar att vi har ett program som skall visualisera grafik i 3D. Vi använder i detta program den klass för att representera vektorer som presenterades i Kapitel 21. Vi kan dock för vårt exempels skull anta att man kallat klassen `vector` istället för `Vector`. Programmet använder sig även av normala vektor-containers från STL (se Kapitel 24) för att lagra våra matematiska vektorer. Vad händer om vi försöker göra kod enligt följande:

```
#include <vector>

class vector {
public:
  vector () {};
};

int main () {
  // skapa en vektor av vektorer
  vector<vector> VektorContainer;
}
```

Exemplet ovan är en aning överdrivet, men det kommer inte att kunna kompileras. Kompilatorn kommer att klaga på något i stil med att `vector` definieras om på rad 3. Vi kan således inte ha två olika klasser med samma namn och använda dem i samma fil. De enda lösningen på problemet med våra kunskaper hittills är att döpa om vår egna matematiska `vector`-klass till något annat som inte kolliderar med klassen från STL.

Vad skulle ha hänt ifall matematik-`vector` skulle komma från ett bibliotek till vilket vi inte har tillgång till källkoden och kan ändra på klassens namn? I så fall har vi två klasser `vector` och vi kan inte ändra på namnet på någon av dem. Lösningen här är att utnyttja namespaces.

25.1.2. Namespacet `std`

Det mesta av de bibliotek som tillhör C++ finns logiskt organiserat i ett enda namespace som heter `std` (förkortning av `standard`). Detta gäller alla klasser som har att göra med t.ex. input/output och STL. På så sätt finns alla standard-klasser och definitioner under "ett tak" och man kan undvika kollisioner med egna eller tredje parts klasser och definitioner.

25.2. Använda namespaces

För att använda en symbol från ett namespace placerar man namnet på namespacet följt av `::` före symbolen då den används. Allmänt:

`namespacenamn``::`*`symbolnamn`*

Vi kan alltså placera `std::` före alla klasser som vi använder som tillhör standarden för C++. Vi kunde skriva ett avancerat "Hello World"-program på följande sätt:

```
#include <iostream>

int main () {
  int Ar;
```

```
  // läs in ett år
  std::cout « "Skriv ett år: ";
  std::cin » Ar;

  // skriv vår hälsning
  std::cout « "Hello world, detta är år " « Ar « endl;
}
```

På de flesta system behöver man inte explicit ange att man vill använda klasser och andra definitioner från namespacet `std`, vilket gäller det system som detta kompendium utvecklats på, men vissa system som bättre följer standarden för C++ kräver det. På ett sådant system kompilerar ovenastående program inte om man lämnar bort prefixet `std::` från `cin` och `cout`.

I alla program som vi hittills skrivit i detta kompendium borde vi använda oss av namespaces. Om den kompilator du använder dig av inte kompilerar exempelprogrammen kan du försöka placera `std::` före t.ex. `cin`, `cout` o.dyl, eller använda syntaxen som presenteras nedan för att alltid använda `std`. I framtida versioner av kompendiet använder kanske alla exempelprogram sig av namespaces på ett helt standardenligt sätt.

25.2.1. Alltid använda ett namespace

Genom att skriva ett namespacenamn följt av `::` kan vi explicit ange att vi vill ha en viss symbol från ett visst namespace, men det kan bli jobbigt i längden. Det är enkelt att glömma att placera `std::` före `cin` och `cout` varje gång de används. Man kan istället definiera att ett visst namespace alltid skall användas, d.v.s. alla symboler skall kunna användas utan att behöva skriva ett prefix före varje symbol. Detta görs med deklarationen `using namespace` som används allmänt enligt följande:

```
using namespace namespacenamn;
```

Om denna deklaration förekommer t.ex. i början på en fil kan man i den filen och alla som inkluderar denna använda symboler från *namespacenamn* utan att tvingas skriva ett prefix. Deklarationen säger att vi nu framöver använder oss av allting som finns i det givna namnspacet. Vi kan skriva om programmet från ovan enligt:

```
#include <iostream>

using namespace std;

int main () {
  int Ar;

  // läs in ett år
  cout « "Skriv ett år: ";
  cin » Ar;

  // skriv vår hälsning
  cout « "Hello world, detta är år " « Ar « endl;
}
```

Vi kan placera en `using`-sats i princip var som helst i en programfil där man normalt kan ha andra satser. I normala fall placerar man dock `using`-satser i början av en fil.

Man kan naturligtvis `using`-deklarera flera olika namespaces. Detta är praktiskt om de inte innehåller namnkollisioner av något slag.

25.2.2. Använda subset av namespace

I alla fall kanske man inte vill "importera" alla symboler som finns i ett namespace, utan endast några få, och använda resten via ett prefix. Använder man satsen `using namespace` *namn*`;` får man ju med alla symboler i den globala symbolrymden. Man kan dock istället deklarera att endast vissa symboler används. Allmänt skriver man då:

```
using namespacenamn::symbolnamn;
```

Ett exempel kan vara att vi vill kunna använda klassen `string` från namespacet `std` utan prefixet `std::`, men andra klasser bör bibehålla prefixet. Vi kan då skriva:

```
using std::string;
```

I ett sammanhang kan det se ut på följande sätt:

```
#include <iostream>
#include <string>

int main () {
  // använd string från namespacet std
```

```
  using std::string;

  string Ord;

  // läs in ett år
  std::cout « "Skriv ett ord: ";
  std::cin » Ord;

  // skriv vår hälsning
  std::cout « "Du skrev in: " « Ord « endl;
}
```

Genom att explicit välja symboler kan man även reda ut namnkollisioner. Man kan då precisera vilken version av någon viss klass som avses då klassen används.

25.2.3. Klasser som namespaces

Notera att då man implementerar metoder för klasser utanför själva `class`-definitionen använder man samma syntax som vid namespaces. Om vi t.ex. implementerar en metod `print()` för en klass `Image` ser det ju t.ex. ut på följande sätt:

```
void Image::print () {
  ...
}
```

Här säger vi explicit att vi implementerar en metod `print()` som tillhör ett namespace `Image`. Klasser fungerar alltså även som namespaces, eftersom de associerar ett antal metoder och data med ett namn (klassens namn).

25.3. Skapa egna namespaces

Man kan förstås skapa egna namespaces för att gruppera kod om man behöver. Detta blir vanligtvis dock aktuellt först då man har stora projekt, eller då man har en namnkonflikt orsakad av bibliotek man inte kan påverka. Moderna C++-bibliotek torde börja introducera namespaces för att undvika denna typ av problem, men alla har det ännu inte.

För att specificera att kod skall tillhöra ett visst namespace kan man göra på följande sätt:

```
namespace namespacenamn {
    deklarationer och definitioner
}
```

Det enda som behövs är att säga var namespacet börjar, vad det heter, samt var det slutar. All kod som skall bli innanför namespacet kan lämnas orörd. Vi kan nu skapa ett eget namespace för att placera vår klass för matematiska vektorer. Vi kallar namespacet för `math`. Man kan namnge namespaces enligt samma regler som gäller för variabler, funktioner, klasser m.m. Se Avsnitt 2.2.1.

```
#include <vector>

// definiera ett nytt namespace
namespace math {

  class vector {
  public:
    vector () {};
  };
}

int main () {
  // skapa en vektor av vektorer
  std::vector<math::vector> VektorContainer;
}
```

Vår egna klass `vector` tillhör nu namespacet `math`, medan STL-klassen `vector` tillhör som normalt `std`. Vi kan genom att explicit ange vilken implementation av `vector` vi avser skapa en `vector` (container) av `vector` (matematiska vektorer).

Vi har i vårt exempel ovan endast en enda klass i vårt namespace, men vi kan inkludera i princip hur många definitioner som helst i ett och samma namespace. Vi kan lägga till en hypotetisk klass `matrix` på följande sätt:

```
#include <iostream>

// definiera ett nytt namespace
namespace math {

  // en matematisk vektor
  class vector {
  public:
```

```
    vector () {};
  };

  // en matematisk matris
  class matrix {
  public:
    matrix (int X, int Y) : m_X(X), m_Y(Y) {};
  private:
    int m_X, m_Y;
  };
}

int main () {
  // skapa en vektor av vektorer
  std::vector<math::vector> VektorContainer;

  // skapa en matrix
  math::matrix M ( 3, 3 );
}
```

Vi kan även placera `typdedef`-definitioner, enumereringar, vanliga funktioner m.m. i namespaces.

25.3.1. Namespaces och prototyper

I de fall vi tittat på ovan har all kod skrivits direkt in i namespacet. Detta fungerar bra för relativt små klasser, men om man vill presentera gränssnittet för t.ex. en stor klass och placera koden separat har vi två alternativ. Vi kan antingen *addera kod till namespacet* eller explict ange till vilket namespace (och klass) en funktion/metod hör. Vi skall se på ett exempel där vi har tre funktioner som vi vill presentera i en headerfil:

```
#ifndef NAMESPACE5_H
#define NAMESPACE5_H

#include <string>

namespace StringManip {
  // definiera prototyper
  std::string toString (int Tal);
  int         toInt    (std::string Text);
  float       tofloat  (std::string Text);
}

#endif
```

Vi har nu definierat prototyper för tre funktioner, samt deklarerat att de tillhör namespacet `StringManip`. Vi har dock inte ännu visat koden för dessa, utan den finns i en separat fil. Hur kan vi då där specificera att vi implementerar funktionerna `toString()`, `toInt()` och `toFloat()` för namespacet `StringManip`? Om vi inte gör det kommer kompilatorn inte att kunna koppla samman prototyperna med den egentliga implementationen, och vi får ett länkningsfel. Det lättaste sättet är att kapsla in implementationen i en likadan namespace-deklaration som prototyperna. Vi kan då få någonting i stil med:

```
#include "Namespace5.h"

namespace StringManip {

  std::string toString (int Tal) {
    // koden bortlämnad
  }

  int toInt (std::string Text) {
    // koden bortlämnad
  }

  float tofloat (std::string Text) {
    // koden bortlämnad
  }
}
```

Funktionernas kod är i detta sammanhang irrelevant och bortlämnad. Detta exempel visar att namespaces är *öppna*, d.v.s. man kan lägga till kod till ett existerande namespace genom att helt enkelt kapsla in koden innanför ett namespace med samma namn. Notera att vi måste se till att vi exakt matchar våra implementationer mot prototyperna, precis som normalt, annars får vi länkningsproblem. Då man använder denna "öppna" syntax är det mycket enkelt att sprida ut koden som tillhör ett visst namespace över många filer utan problem.

Den andra metoden att lägga till kod till ett namespace innebär att varje symbol ges ett prefix som är namnet på namespacet som symbolen skall placeras i. Vi kan då skriva funktionerna ovan på följande sätt:

```
#include "Namespace5.h"

std::string StringManip::toString (int Tal) {
  // koden bortlämnad
}
```

```
int StringManip::toInt (std::string Text) {
  // koden bortlämnad
}

float StringManip::tofloat (std::string Text) {
    // koden bortlämnad
}
```

Varje funktion har nu prefixet `StringManip::`, som anger att funktionen i fråga tillhör namespacet `StringManip`. Notera att det är samma syntax som används för att specificera vilken klass en metod tillhör. Här kan många likheter märkas, eftersom en klass även kan ses som en form av namespace, och dess metoder och definitioner är kapslade "under samma tak". Vad händer då om vi har en klass som tillhör ett namespace och vars metoder vi vill implementera separat, kanske i en skild fil? En enkel lösning är ju att skriva metoderna som normalt och sedan kapsla in dem innanför ett enda namespace, såsom vi gjorde i den första versionen tidigare i detta avsnitt. En annan lösning är att använda ett prefix, precis som vi gjorde ovan. Vi kan ta som exempel en klass `Coordinate` som vi lägger till till namespacet `math`:

```
#include <iostream>

namespace math {
  class Coordinate {
  public:
    // konstruktor
    Coordinate (float X, float Y);

    // accessera medlemmar
    float x ();
    float y ();
    void setX (float X);
    void setY (float Y);

  private:
    // medlemmar
    float m_X, m_Y;
  };
}

// implementera alla metoder.
math::Coordinate::Coordinate (float X, float Y) { m_X = X; m_Y = Y; };
float math::Coordinate::x () { return m_X; };
float math::Coordinate::y () { return m_Y; };
void  math::Coordinate::setX (float X) { m_X = X; };
void  math::Coordinate::setY (float Y) { m_Y = Y; };
```

```
int main () {
  using namespace std;

  math::Coordinate C (1,1);
  std::cout « "Koordinater: (" « C.x () « ","
            « C.y () « ")" « endl;
}
```

Vi har här skrivit metoderna helt normalt, med den enda skillnaden att de fått ett extra prefix `math::`. Implementationen av metoderna kan bli ganska svårläst då man använder denna metod, eftersom det blir ett extra prefix, men istället vet läsaren direkt vilket namespace (och förstås även vilken klass) metoden tillhör. Vilken metod man väljer för egen kod spelar ingen roll.

25.4. Aliaser

Man kan definiera s.k. *aliaser* (smeknamn) för namespaces. Detta kan vara praktiskt om ett namespace har ett långt och klumpigt namn som är jobbigt att använda. Vi kan t.ex. anta att institutioner för informationsbehandling har en samling klasser som skall användas i olika kurser, och vill samla alla dessa klasser i ett och samma namespace för att undvika problem för användarna. Detta namespace kallas `DepartmentOfComputerScience`, vilket ju är ganska långt och klumpigt, och en alias kan därför vara på sin plats. Vi kan t.ex. ha något i stil med:

```
namespace DepartmentOfComputerScience {
  class Foo {
  public:
    Foo () {}
  };
}

int main () {
  // använd namespacet oförkortat
  DepartmentOfComputerScience::Foo F1;

  // definiera en alias
  namespace CS = DepartmentOfComputerScience;

  // använd aliasen
  CS::Foo F2;
```

```
}
```

Här har vi definierat en alias som heter `CS` för det långa namnet `DepartmentOfComputerScience`. Aliaser skapas allmänt enligt följande:

```
namespace alias = namespacenamn;
```

Man kan använda både originalet och aliasen helt parallellt, aliasen är endast ett annat namn för samma namespace.

25.5. Diskussion

Vi har här visat att namespaces är ett mycket praktisk sätt att gruppera kod i moduler på en högre nivå en klasser. Namespaces kommer till sin rätt närmast då man har större helheter kod som skall implementeras, och risken för kollisioner mellan klasser coh definitioner är överhängande.

Det är även rekommendabelt att definiera ett eget namespace ifall man arbetar med ett projekt som resulterar i ett *bibliotek* som skall användas av andra personer. I så fall kan andra användare lättare undvika nanproblem som bibloteket annars kunde åstadkomma. Det är inte svårt eller arbetsdrygt att definiera ett namespace, så det lönar sig.

Kapitel 26. Typinformation

Detta kapitel behandlar ett ganska nytt tillägg till standarden för C++, nämligen möjligheten att kunna få reda på ett objekts typ dynamiskt.

26.1. Varför dynamisk typinformation

Dynamisk typinformation kan vara praktisk att ha då man vill veta exakt vilken typ av objekt man hanterar just då. Om vi har en klasshierarki med klasserna `A`, `B` och `C` som är definierade enligt:

```
class A { ... };
class B : public A { ... };
class C : public B { ... };
```

Klassen `B` har några metoder som inte `A` har och `C` har några metoder som inte `B` har. Vi kan även anta att vi har en t.ex. en metod som tar som parameter en pekare till ett objekt av typen `A`. Objekten som egentligen skickas till metoden kan vara av alla tre typer. Vi utför någon viss virtuell metod som är definierad för alla objekt som skickas, men vi skulle även vilja exekvera någon extra metod för objekten av typen `B` och `C`. Det enda som vi hittills har kunnat göra är att t.ex. definiera en metod i `A` som returnerar objektets typ, t.ex. en sträng. Det är dock väldigt anti-OO, och därför finns det funktioner för att utföra dynamisk konvertering av objekt till en annan klass (om det är möjligt).

Typkonvertering fungerar *endast* med klasser som har *virtuella funktioner*. Dessa typer av klasser är även de enda typerna av klasser där man skall referera till ett objekt via en pekare till en basklass.

26.2. Hur används typkonvertering

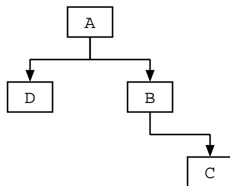
Funktionen som används för konverteringen heter `dynamic_cast` och tar som parameter en datatyp till vilken man skall försöka konvertera och en pekare till ett

objekt. Allmänt används `dynamic_cast` på följande sätt:

```
datatyp * variabel = dynamic_cast<datatyp *>(objekt);
```

Man placerar den datatyp till vilken man vill försöka konvertera ett objekt innanför <>, precis som då man använder templates (se Kapitel 23). En pekare till det objekt som man vill konvertera ges som parameter. `dynamic_cast` returnerar en pekare till konverterat objekt ifall konverteringen var möjlig och 0 ifall det inte var möjligt. Notera att pekaren ännu pekar på samma objekt, men det bara "tolkas" som ett objekt av en annan datatyp. Innehållet är dock detsamma. Vad kan man då konvertera? Man kan konvertera en pekare till en basklass till en pekare till en subklass, förutsatt att objektet från början skapats som ett objekt av subklassen. Man kan dock inte konvertera ett objekt till ett annat om det inte är en super- eller subklass. Man kan inte heller konvertera ett objekt som är skapat som ett objekt av en basklasstyp till en subklasstyp. Vi ser på den arvshierarki vi hade ovan och lägger till en klass `D` som ärver `A`:

Figur 26-1. Arvshierarki



Anta att vi har ett objekt som är skapat med följande anrop:

```
A * BasKlass = new A;
```

Detta objekt kan inte konverteras till någon annan klass. Om vi istället har följande objekt:

```
C * SubSub = new C;
```

och vi refererar till det från en metod som tagit een pekare till ett objekt av typen `A` som argument, så kan vi konvertera det till både `B` och `C`. Detta går eftersom ett objekt av typen `C` är (*is-a*) även ett objekt av type `B` och ett av typen `A` (se Kapitel 15).

Om vi däremot har ett objekt av typen `D` som refereras via en pekare av typen `A` kan det *inte* konverteras till varken `B` eller `C`, eftersom ingendera klassen är relaterad till `D` på något annat sätt än att de har samma basklass `A`.

26.2.1. Praktiskt exempel

Vi kan definiera en metod som tar objekt av typen `A` på följande sätt:

```
void doIt (A * Data) {
  C * ObjC;
  B * ObjB;

  // försök konvertera till ett 'C'
  if ( ( ObjC = dynamic_cast<C *>(Data) ) != 0 ) {
    cout « "Objektet av typen C" « endl;
  }

  // ett 'B'?
  else if ( ( ObjB = dynamic_cast<B *>(Data) ) != 0 ) {
    cout « "Objektet av typen B" « endl;
  }

  // måste vara ett 'A'
  else {
    cout « "Objektet av typen A" « endl;
  }
}
```

Funktionen använder sig av `dynamic_cast` för att försöka konvertera till de olika subklasserna. Man måste börja med den mest ärvda klassen, annars får man kanske inte det man vill ha. Om vi t.ex. skickar ett objekt av typen `C` vill vi att det skall känas igen som ett `C` och ingenting annat. Men om vi testar om objektet är av typen `B` *före* vi testar om det är av typen `C` så kommer det att kunna konverteras till ett `B` (det *är* ju trots allt en subklass av `B`!), och således testas inte alls omd et är av typen `C`. Vi kunde ju förstås lämna bort `else`-delarna ovan så skulle det problemet vara löst.

26.3. Typinformation

Man kan även få information från en datatyp om vilken typ det är frågan om. I normala fall vet man vilken datatyp det är då det är frågan om primitiva datatyper, men vid arvshierarkier och med pekare till basklasser kan man inte alltid vara säker. Man kan i sådana fall använda sig av t.ex. någon inbyggd metod eller så `dynamic_cast` för att försöka konvertera objektet till någon viss klass. Ett alternativt sätt är att använda en annan ny finess hos C++, man kan nämligen få ut en *id* för varje objekt som säger vilken datatyp det är frågan om. Denna id kan användas för att jämföra om två objekt är av samma typ. Funktionen som ger informationen heter `typeid()` och returnerar en referens till en klass `const type_info`. Denna funktion kan användas efter att headerfilen `<typeinfo>` inkluderats. Ett exempel på hur `typeinfo()` kan användas:

```
#include <typeinfo>

void foobar (A * Data) {
  // hämta information om 'Data'
  const type_info & Info = typeid ( *Data );
  cout « "Typen = " « Info.name () « endl;
}
```

Vi måste ha en referens till en `type_info` för returvärdet från `typeid()`. Notera att funktionen returnerar typen på dess parameter, så skickar vi t.ex. en pekare får vi ut som information att det är en pekare till någon datatyp, därför avrefereras `Data` innan den skickas som parameter. Klassen `type_info` har överlagrade operatorer (se Kapitel 21) som tillåter jämförelse (`==` och `!=`) mellan olika instanser av `type_info`. En metod `name()` finns tillgänglig ifall man vill använda en strängrepresentation av datatypen till något, t.ex. till en `map` över olika klasser.

26.4. Missbruk av typinformation

Alltför ofta ser man olika program missbruka möjligheten att få ut information om olika datatyper. Programmerare behöver inte tänka på sina klashierarkier, eftersom man ändå alltid kan kolla vilken typ av objekt man hanterar vid en given tidpunkt. I många fall används t.ex. `typeid()` för att utföra olika metoder på bas av vilken typ av objekt

man just då har. Detta kunde lika väl göras med en virtuell metod som är överlagrad i de olika klasserna. Det finns för det mesta alternativ till att använda dynamisk typinformation. I vissa fall är denna information bra, men den får inte överanvändas och missbrukas. Även typinformation är något som purister inom objektorientering skulle vilja stryka ur standarden för C++.

Appendix A. C-strängar

Detta appendix behandlar hur man använder sig av strängar i C, utan att använda klassen `string`.

A.1. Varför C-strängar?

Vi skall behandla C-strängar kort och koncist eftersom det ännu finns en enorm mängd program som använder sig av C-strängar, och som troligtvis aldrig kommer att övergå till att använda `string`. Även många nya program som görs idag använder sig av C-strängar, trots att `string` skulle göra programmet både lättare att utveckla och även minska antalet strängrelaterade minnesfel. De flesta olika bibliotek som finns idag använder även sig av C-strängar, t.ex. olika systemanrop. Det är därför bra att ha en bild av hur de fungerar, vilka operationer som kan utföras och vilka som inte kan utföras.

A.2. Hur C-strängar fungerar

Ser man på hur strängar i C fungerar kan man säga attd et egentligen är en *nollterminerad vektor av* `char`. Med nollterminerad menas att varje sträng bör sluta med värdet 0. Här avses inte siffran ’0’, utan ASCII-värdet 0, som inte är ett giltigt tecken som kan visas på skärmen. Detta tecken anger var strängen slutar, och de flesta funktioner som använder sig av C-strängar är beroende av att det finns på korrekt plats.

Figur A-1. Representation av C-sträng

H	e	l	l	o	!	\0

Om inte ett tecken 0 terminerar strängen kan allvarliga fel ske då funktioner som blint

väntar sig ett 0 inte hittar det utan fortsätter ut i okänt minne. Sådana fel är inte helt sällsynta. C-strängar är därför alltid ett tecken längre än vad antalet egentliga tecken i strängen kräver. I bilden ovan finns det sex tecken som kan visas på t.ex. skärmen, men strängen är ändå sju tecken lång för att rymma en sista 0.

A.2.1. Skapa C-strängar

Eftersom C-strängar är helt enkelt vektorer med element av typen `char` kan man skapa strängar precis som `char`-vektorer, t.ex.:

```
char Hello1[] = "Hello!";
char * Hello2 = "Hello again!";
char Hello3[5];

Hello3[0] = 'H';
Hello3[1] = 'e';
Hello3[2] = 'j';
Hello3[3] = '!';
Hello3[4] = '\0';
```

Den första variabeln `Hello1` definierar en vektor utan storlek, och låter kompilatorn räkna ut denna. Strängen blir exakt så lång som det behövs för alla tecken och terminatorn 0, d.v.s. sju tecken. `hello2` däremot använder en alternativ syntax, men som åstadkommer exakt samma reslutat. Med `hello3` skapas en vektor med en explict storlek. Vi fyller sedan manuellt vektorn genom att ge värden åt de enskilda tecknen, utan att glömma det sista tecknet `'\0'`. Då man definierar en explicit storlek måste man vara mycket noggrann så att den tilltänkta strängen ryms i vektorn.

Kan man naturligtvis även dynamiskt allokera C-strängar. Man allokerar då en vektor av datatypen `char` med den önskade storleken.

A.2.2. Manipulera enskilda tecken

Man kan manipulera de enskilda tecknen i C-strängar precis som element av typen `char` i vilken vektor som helst. I exemplet ovan tilldelades en sträng ett värde genom elementvis tilldelning. Man kan accessera elementen normalt genom operatorn `[]` eller med hjälp av pekararitmetik. Vi kunde skriva om ovanstående tilldelning till följande:

```
char Hello3[5];

*(Hello3 + 0) = 'H';
*(Hello3 + 1) = 'e';
*(Hello3 + 2) = 'j';
*(Hello3 + 3) = '!';
*(Hello3 + 4) = '\0';
```

I vissa fall är det lämpligt att använda pekararitmetik, men i de flesta fall är det lättare och snyggare att använda normal vektorindexering med `[]`. Det är ingen skillnad ur effektivitetssynpunkt, eftersom kompilatorn översätter allting till pekararitmetik innan den egentliga koden genereras. Använd den syntax som verkar lättare och som ger mindre fel.

A.3. Tilldelning och jämförelse

Här börjar C-strängar skilja sig från `string`. Varken tilldelning eller jämförelse med C-strängar är speciellt intuitiv, och man behöver för det mesta ta extra funktioner till hjälp.

A.3.1. Tilldelning

Strängar kan inte direkt tilldelas. Eftersom det är frågan om vektorer och således även pekare är det enda som händer att pekarens värde kopieras och båda pekarna refererar ännu till samma minnesområde och sträng. Exempel:

```
#include <iostream>

int main () {
  char Hello1[] = "Hello 1!";
  char * Hello2;

  // skriv ut Hello1
  cout « "Hello1 = '" « Hello1 « "'" « endl « endl;

  // tilldelning
  Hello2 = Hello1;
```

```
  // manipulera Hello2
  Hello2 [6] = '2';

  // skriv ut Hello1 och Hello2
  cout « "Hello1 = '" « Hello1 « "'" « endl;
  cout « "Hello2 = '" « Hello2 « "'" « endl;
}
```

Strängen `Hello2` blir inte en kopia, utan den pekar ännu på samma omtåde som `Hello1`. Följdaktligen påverkar manipuleringen av ett tecken i `Hello2` även `Hello1`, vilket visas i utskriften. Där har teckent 1 bytts till 2 i båda strängarna. Vi har gjort en *grund kopia* (shallow copy) av den ursprungliga strängen. I grunda kopior påverkas originalet. Vad vi egentligen för det mesta vill åstadkomma är *djupa kopior* (deep copy) där hela strängens värde kopieras. Detta kan åstadkommas med t.ex följande kod:

```
char Hello1[5] = "Hej!";
char Hello2[5];

// utför kopieringen teckenvis
for ( int Index = 0; Index < 5; Index++ )
  Hello2[Index] = Hello1[Index];
```

Notera att här har vi färdigt allokerat minne för `Hello2`. Man måste alltid försäkra sig om att det finns allokerat minne dit man ämnar kopiera en sträng. Mera om kopiering i Avsnitt A.5.

A.3.2. Jämförelse

Jämförelser kan inte direkt utföras med operatorn `==` eller `!=`. Inte heller kan man utföra alfabetisk jämförelse med `<`, `<=`, `>=` eller `>`. Vi måste jämföra teckenvis. Om de jämförda strängarna är lika långa och innehåller exakt samma tecken är de lika. Vi kan skriva en egen jämförelsefunktion på följande sätt:

```
#include <iostream>

bool compare (const char * S1, const char * S2) {
  int Index1 = 0;
  int Index2 = 0;

  // säkerhet
  if ( ! S1 || ! S2 )
// någondera (eller båda) strängarna odefinierade
```

```
return false;

  while ( S1[Index1] != '\0' && S2[Index2] != '\0' ) {
// är tecknen ännu lika?
if ( S1[Index1++] != S2[Index2++] )
  // nope, så strängarna är olika
  return false;
  }

  // strängarna är lika
  return true;
}

int main () {
  char * S1 = "Hej!";
  char * S2 = "Hello world, this is main() calling";
  char * S3 = "Hello world, this is main() calling";

  // jämförelse
  if ( compare ( S1, S2 ) )
cout « "S1 och S2 är lika" « endl;
  else
cout « "S1 och S2 är olika" « endl;

  if ( compare ( S2, S3 ) )
cout « "S2 och S3 är lika" « endl;
  else
cout « "S2 och S3 är olika" « endl;
}
```

Körning ger följande resultat:

```
% ./CStringCompare1
S1 och S2 är olika
S2 och S3 är lika
%
```

Jämförelsen är alltså ganska trivial till sin natur att implementera, men det finns en färdig funktion som man kan använda, som är en del mångsidigare. Den heter `strcmp()` och ser allmänt ut på följande sätt:

```
#include <string.h>
int strcmp(const char * S1, const char * S2);
```

Den finns definierad i `<string.h>`. Returvärdet är en aning speciellt, och är:

- `< 0` om *S1* är mindre än `S2`.

- `== 0` om *S1* är lika med `S2`.
- `> 0` om *S1* är större än `S2`.

Man kan på så sätt enkelt utföra alfabetisk jämförelse mellan två strängar. Funktionen antar att båda strängarna är korrekt terminerade av `'\0'`. Vi kan då skriva om vårt program från ovan på följande sätt:

```
#include <iostream>
#include <string.h>

int main () {
  char * S1 = "Hej!";
  char * S2 = "Hello world, this is main() calling";
  char * S3 = "Hello world, this is main() calling";

  // jämförelse
  if ( strcmp ( S1, S2 ) == 0 )
cout « "S1 och S2 är lika" « endl;
  else
cout « "S1 och S2 är olika" « endl;

  if ( strcmp ( S2, S3 ) == 0 )
cout « "S2 och S3 är lika" « endl;
  else
cout « "S2 och S3 är olika" « endl;
}
```

Vi är här endast intresserade av om de är lika, inte hur de förhåller sig alfabetiskt till varandra.

En varient på funktionen ovan tar som ett tredje argument antalet tecken som maximalt skall jämföras. Den heter `strncmp()`. Vi kan modifiera programmet ovan en aning:

```
#include <iostream>
#include <string.h>

int main () {
  char * S1 = "Hello world";
  char * S2 = "Hello";

  // jämförelse
  if ( strncmp ( S1, S2, 5 ) == 0 )
cout « "S1 och S2 är lika" « endl;
  else
cout « "S1 och S2 är olika" « endl;
}
```

Körning ger nu resultatet:

```
% ./CStringCompare2
S1 och S2 är lika
%
```

Eftersom vi jämförde endast fem tecken av de båda strängarna gav `strncmp()` svaret att de är lika. Med normala `strcmp()` eller ett längre intervall till `strcmp()` hade strängarna varit olika.

A.4. Längden av strängar

Man kan relativt enkelt beräkna längden av en sträng med t.ex. följande funktion:

```
#include <iostream>

int length (const char * String) {
  int Len = 0;

  if ( ! String ) {
    // ingen sträng givet
    return 0;
  }

  // iterera tills vi stöter på terminatorn
  while ( String[Len] != '\0' ) {
    Len++;
  }

  return Len;
}

int main () {
  char * S1 = "Hej!";
  char * S2 = "Hello world, this is main() calling";

  cout « S1 « " är " « length (S1) « " tecken lång." « endl;
  cout « S2 « " är " « length (S2) « " tecken lång." « endl;
}
```

Ovanstående funktion kan hantera alla strängar så länge som de är terminerade av teckent `'\0'`. Den kan även kontrollera ifall en odefinierad pekare (nollpekare)

skickats som parameter, och returnerar i så fall 0. Istället för att varje gång skriva en dylik funktion kan vi använda en standard funktion för att beräkna längden. Den heter `strlen()` och tar som argument en `const char *` och returner en `size_t` (positivt heltal). Denna funktion finns definierad i en headerfil `<string.h>` som även innehåller en del andra C-strängrelaterade funktioner. Notera extensionen `.h`. Blanda inte ihop denna med headerfilen `<string>` som ju defineirar normala `string`. De är tyvärr väldigt lika. Ovanstående program kan skrivas kortare med `strlen()`:

```
#include <iostream>
#include <string.h>

int main () {
  char * S1 = "Hej!";
  char * S2 = "Hello world, this is main() calling";

  cout « S1 « " är " « strlen (S1) « " tecken lång." « endl;
  cout « S2 « " är " « strlen (S2) « " tecken lång." « endl;
}
```

A.5. Kopiering av strängar

Då man vill kopiera strängar skiljer sig C-strängar mycket från `string`. Kopieringen kan relativt enkelt gå fel om man inte är försiktig. Orsaken till detta är att man måste manuellt göra all minneshantering och gör amn ett fel där kan hela programmet krascha. Vi kan göra en enkel funktion som utför en kopiering:

```
#include <iostream>

void copy (char * Source, char * Destination) {
  // iterera över och kopiera alla tecken
  for ( unsigned int Index = 0; Index <= strlen ( Source ); Index++ ) {
    // kopiera ett tecken
    Destination[Index] = Source[Index];
  }
}

int main () {
  char Hello1[5] = "Hej!";
  char Hello2[5];

  // utför kopieringen
  copy ( Hello1, Hello2 );
```

```
  cout « "Hello1='" « Hello1 « "', Hello2= '" « Hello2 « '"' « endl;
}
```

Notera att vi i funktionen `copy()` kopierar ett tecken mera än vad strängen är lång. Detta görs för att få med den sista terminerande `\0`, annars är strängen oterminerad, vilket leder till provlem senare. Vi gör heller ingen som helst kontroll på att `Destination` kan rymma alla tecken som finns i `Source`. Detta är upp till anroparen att kontrollera. Vad händer om vi försöker kopiera in i en för kort sträng? Vi modifierar programmet ovan så att det utför en felaktig kopiering:

```
int main () {
  char Hello1[] = "Hej! Detta är en lååång sträng.";
  char Hello2[1];

  // utför kopieringen
  copy ( Hello1, Hello2 );

  cout « "Hello1 = '" « Hello1 « "', " « endl
       « "Hello2 = '" « Hello2 « "'" « endl;
}
```

Det använder samma funktion `copy()` som ovan. Resultatet av körningen blev på den använda maskinen:

```
% ./CStringCopy3
Hello1 = 'ej! Detta är en lååång sträng.',
Hello2 = 'Hej! Detta är en lååång sträng.'
%
```

Notera att `Hello1` har tappat det första H:et. Men vid olika körningar och med olika längder på `Hello2` hände antingen ingenting eller så kraschade programmet med en `segmentation fault`. Detta illustrerar att det kan fungera helt fint att köra ett program som hanterar strängar fel, och felet kan visa sig på helt andra ställen än där det gjordes. Orsaken här är att `Hello2` delvis skriver över `Hello1`, d.v.s. att i minnet finns `Hello2` direkt före `Hello1` och endast det första tecknet (ett "H") hamnar in i `Hello2` och resten kommer i `Hello1` (började från "e"). I vårt fall var det lätt att hitta felet, men tänk om vi istället hade korrupterat någon annan helt orelaterad variabel eller objekt. Dylika fel kan vara mycket svåra att hitta.

En bättre version av ovanstående kan göras ifall vi manuellt allokerar så mycket minne som behövs för den nya strängen samt dess terminator. Vi kan i så fall skriva om `main()` från ovan så att det blir:

```
int main () {
  char Hello1[] = "Hej! Detta är en lååång sträng.";
  char * Hello2;

  // allokera minne
  Hello2 = new char [ strlen ( Hello1 ) + 1 ];

  // utför kopieringen
  copy ( Hello1, Hello2 );

  cout « "Hello1 = '" « Hello1 « "', " « endl
       « "Hello2 = '" « Hello2 « "'" « endl;

  delete [] Hello2;
}
```

Enda skillnade är att `Hello2` är en pekare och inte en vektor och att vi allokerar så mycket minne som behövs före kopieringen. Detta är det korrekta sättet att utföra en kopiering.

A.5.1. Färdiga funktioner

Det finns några färdiga funktioner som gör livet lättare då man vill kopiera C-strängar. Den ena och mycket använda funktionen är `strcpy()`, som kopierar en sträng till en annan. Den fungerar på samma sätt som vår funktion `copy()` med de enda skillnaderna att den även returnerar destinationssträngen och att parametrarna är omsvängda. `strcpy()` finns definierad i `<string.h>`. Även här måste man försäkra sig om att destinationssträngen innehåller tillräckligt med utrymme för hela den kopierade strängen, annars får man liknande fel som med `copy()`. Vi kan skriva om vårt kopierande program från ovan:

```
#include <string.h>

int main () {
  char Hello1[] = "Hej! Detta är en lååång sträng.";
  char * Hello2;

  // allokera minne
```

```
  Hello2 = new char [ strlen ( Hello1 ) + 1 ];

  // utför kopieringen
  strcpy ( Hello2, Hello1 );

  cout « "Hello1 = ’" « Hello1 « "’, " « endl
       « "Hello2 = ’" « Hello2 « "’" « endl;

  delete [] Hello2;
```

I övrigt är `strcpy()` enkel att använda. Det finns en alternativ version av `strcpy()` som heter `strncpy()` som kopierar ett visst antal tecken från en sträng. Där måste man dock vara noggrann med att man manuellt ser till att terminatorn hamnar på rätt plats, eftersom `strnncpy()` inte automatiskt placerar en dylik i destinationssträngen. Vi kan använda oss av `strncpy()` för att kopiera substrängar.

```
#include <iostream>
#include <string.h>

int main () {
  char Hello1[] = "Hej! Detta är en lååång sträng.";
  char * Hello2;

  // allokera minne
  Hello2 = new char [ strlen ( "Detta" ) + 1 ];

  // utför kopieringen
  strncpy ( Hello2, Hello1 + 5, 5 );
  Hello2[5] = ’\0’;

  cout « "Hello1 = ’" « Hello1 « "’, " « endl
       « "Hello2 = ’" « Hello2 « "’" « endl;

  delete [] Hello2;
}
```

Vi vill här kopiera en substräng som börjar på det femte tecknet i `Hello1` och är fem tecken lång. Vi använder oss av normal pekararitmetik för att ange minnespositionen för det femte tecknet i `Hello1`. Den sista parametern är antalet tecken som skall kopieras. Vi måste här manuellt se till att `Hello2` termineras med `\0` eftersom vi inte kopierade med ett sådant tecken. Notera även att vi kan kontrollera längden på en strängkonstant med hjälp av `strlen()` för att undvika att vi räknar för hand och gör ett fel. Kör man ovanstående program får man denna utskrift:

```
% ./CStringSubstring1
```

```
Hello1 = 'Hej! Detta är en lååång sträng.',
Hello2 = 'Detta'
%
```

En alternativ funktion som kan användas är `strdup()` som duplicerar en sträng. Enda problemet här är att `strdup()` använder sig av `malloc()` för att göra en allokering för den nya strängen, och den allokerade strängen skall således frigöras med `free()`. Mera information om dessa på Avsnitt 12.4. Använder man `strdup()` får man alltså inte använda `delete` för att frigöra minnet. Ett exempel:

```
#include <iostream>
#include <string.h>
#include <stdlib.h>

int main () {
  char * S1 = "Hello world";
  char * S2;

  // utför kopieringen
  S2 = strdup ( S1 );

  cout « "S1 = '" « S1 « "', " « endl
       « "S2 = '" « S2 « "'" « endl;

  // frigör strängen med C:s funktion free()
  free ( S2 );
}
```

Man bör dock undvika `strdup()` för att inte blanda ihop allokeringar gjorda med `malloc()` och `new`.

A.6. Konkatenering av strängar

Konkatenering av C-strängar är en ganska enkel procedur. Man skriver helt enkelt alla tecken från en viss sträng på slutet av en annan. Man kan göra en enkel funktion för konkatenering av två strängar på följande sätt:

```
#include <iostream>

void concat (char * Source, char * Destination) {
  // beräkna slutet på destinationssträngen
```

```
  unsigned int Start = strlen ( Destination );

  // iterera över och kopiera alla tecken
  for ( unsigned int Index = 0; Index <= strlen ( Source ); Index++ ) {
    // kopiera ett tecken
    Destination[Start + Index] = Source[Index];
  }
}

int main () {
  char Destination[100] = "En exempelsträng";
  char Source [] = ", och en annan";

  // utför kopieringen
  copy ( Source, Destination );

  cout « "Source      = '" « Source« "', " « endl
       « "Destination = '" « Destination « "'" « endl;
}
```

Funktionen `concat()` kopierar helt enkelt strängen `Source` till slutet på `Destination` med början från den position där terminatortecknet finns. Ingen kontroll görs att `Destination` är tillräckligt lång. Här gäller samma som för kopiering av strängar (se Avsnitt A.5), d.v.s. att om `Destination` är för kort uppstår olika typer av fel. Kör man programmet ovan får man:

```
% ./CStringConcat1
Source      = ', och en annan',
Destination = 'En exempelsträng, och en annan'
%
```

Även för denna funktionalitet finns det färdiga funktioner i `<string.h>`. Konkatenering kan göras med funktionen `strcat()`. Denna tar samma parametrar som vår `concat`, men i omvänd ordning. Vårt program ovan kunde kortare skrivas:

```
#include <iostream>
#include <string.h>

int main () {
  char Destination[100] = "En exempelsträng";
  char Source [] = ", och en annan";

  // utför kopieringen
  strcat ( Destination, Source );

  cout « "Source      = '" « Source« "', " « endl
       « "Destination = '" « Destination « "'" « endl;
}
```

Vill man vara säker på att den konkatenerade strängen ryms måste man manuellt beräkna längderna på de involverade strängarna och kontrollera att de är tillräckliga.

En version av `strcat()` som endast konkatenerar ett visst antal tecken finns även, och den heter `strncat()`. Denna har en tredje parameter som anger antalet tecken som skall kopieras. Funktionen `strncat()` sätter alltid ett terminatortecken i slutet på `Destination`.

A.7. Andra strängrelaterade funktioner

Andra strängrelaterade funktioner kan man själv skapa genom att direkt manipulera strängars innehåll såsom vi gjort i våra exempelfunktioner. Det finns dock ett antal olika funktioner som gör det lätare att manipilera strängar. Se dock Avsnitt A.8 för mera information om vilka funktioner som finns tillgängliga.

A.7.1. Söka tecken

Sökning av ett viss tecken i en C-sträng kan ganska enkelt implementeras genom att man jämför strängens tecken enskilt tills man hittat det sökta tecknet eller kommit till strängens slut. En enkel version kan skrivas på följande sätt:

```
#include <iostream>

int findChar (char * String, char Seek) {
  int Index = 0;

  // iterera tills vi kommer till slutet
  while ( String[Index] != ’\0’ ) {
// är det korrekt tecken?
if ( String[Index] == Seek ) {
  // vi har korrekt index
  return Index;
}
Index++;
  }

  // inget index hittades
  return -1;
}
```

```
int main () {
  int Index;
  char * S = "En sträng ur vilken vi söker text";

  cout « "Söksträngen är: ’" « S « "’" « endl;

  // sök ett ’v’
  if ( ( Index = findChar (S, ’v’ )) != -1 ) {
cout « "Hittade ett ’v’ på index " « Index « endl;
  }
  else {
  cout « "Hittade inget ’v’" « endl;
  }

  // sök ett ’P’
  if ( ( Index = findChar (S, ’P’ )) != -1 ) {
cout « "Hittade ett ’P’ på index " « Index « endl;
  }
  else {
  cout « "Hittade inget ’P’" « endl;
  }
}
```

Funktionen `findChar()` söker igenom strängen `String` tills antingen slutet av strängen nåtts eller korrekt tecken hittats. Om ett tecken hittas returneras dess index, men om inget tecken motsvarande `Seek` hittas returneras `-1` för att indikera en misslyckad sökning. Körning av programmet ger:

```
% ./CStringFind1
Söksträngen är: ’En sträng ur vilken vi söker text’
Hittade ett ’v’ på index 13
Hittade inget ’P’
%
```

Variationer på denna funktion är funktionerna `index()` och `rindex()` som finns i `<string.h>`. De söker även efter ett tecken ur en sträng, men istället för att returnerna ett index returnerar de en pekare till tecknet, eller `0` om tecknet inte kunde hittas. De returnerar alltså en perkare till den substräng som börjar där tecknet hittades. Skillnaden mellan `index()` och `rindex()` är att den förra söker från början av strängen medan den senare söker från slutet av strängen. Vi kunde således skriva om vårt program från ovan på följande sätt:

```
#include <iostream>
#include <string.h>
```

```
int main () {
  char * Found;
  char * S = "En sträng ur vilken vi söker text";

  cout « "Söksträngen är: '" « S « "'" « endl;

  // sök ett 'v'
  if ( ( Found = index (S, 'v' )) != 0 ) {
cout « "Hittade ett 'v' på index " « Found - S « endl;
  }
  else {
  cout « "Hittade inget 'v'" « endl;
  }

  // sök ett 'P'
  if ( ( Found = index (S, 'P' )) != 0 ) {
cout « "Hittade ett 'P' på index " « Found - S « endl;
  }
  else {
    cout « "Hittade inget 'P'" « endl;
  }
}
```

Notera att vi använder oss av pekaraitmetik för att beräkna indexet i strängen. Vi vet att om tecknet hittades är det längre fram i minnet än `S`, så vi kan subtrahera `S` från `Found` utan problem för att hitta samma index som `findChar()` gav oss. Använda `rindex()` fungerar på samma sätt.

A.7.2. Söka substrängar

Förutom att endas söka efter enskilda tecken i en sträng kan det förekomma situationer då man behöver kontrollera om en viss sträng är en substräng av en annan sträng, och i så fall på vilket index den börjar. Konceptuellt är det ganska enkelt att göra, och vi kan använda oss av följande funktion för att utföra sökningen:

```
#include <iostream>
#include <string.h>

int findSubstring (char * String, char * Seek) {
  unsigned int Index;

  for ( Index = 0; Index <= strlen (String) - strlen (Seek); Index++ ) {
    // kanske strängen börjar här?
    if ( strncmp (String + Index, Seek, strlen (Seek)) == 0 ) {
```

```
      // vi hittade den!
      return Index;
    }
  }

  // ingen sådan sträng
  return -1;
}

int main () {
  int Index;
  char * S = "En sträng ur vilken vi söker text";

  cout « "Söksträngen är: '" « S « "'" « endl;

  // sök efter "text"
  if ( ( Index = findSubstring (S, "text" )) != -1 ) {
    cout « "Hittade 'text' på index " « Index « endl;
  }
  else {
    cout « "Hittade inte 'text'" « endl;
  }

  // sök efter "finns inte"
  if ( ( Index = findSubstring (S, "finns inte" )) != -1 ) {
    cout « "Hittade 'finns inte' på index " « Index « endl;
  }
  else {
    cout « "Hittade inte 'finns inte'" « endl;
  }
}
```

Funktionen `findSubstring()` använder sig av normala `strncmp()` för att kontrollera om substrängen `Seek` börjar någonstans i `String`. Körning av programmet ger:

```
% ./CStringFind3
Söksträngen är: 'En sträng ur vilken vi söker text'
Hittade 'text' på index 29
Hittade inte 'finns inte'
%
```

Vi kan även här använda oss av en existerande funktion ur `<string.h>` för att göra samma som `findSubstring()`. Funktionen heter `strstr()`. Vi kan då skriva om vårt program på följande sätt:

```
#include <iostream>
```

```
#include <string.h>

int main () {
  char * Found;
  char * S = "En sträng ur vilken vi söker text";

  cout « "Söksträngen är: ’" « S « "’" « endl;

  // sök efter "text"
  if ( ( Found = strstr ( S, "text" )) != 0 ) {
cout « "Hittade ’text’ på index " « Found - S « endl;
  }
  else {
  cout « "Hittade inte ’text’" « endl;
  }

  // sök efter "finns inte"
  if ( ( Found = strstr ( S, "finns inte" )) != 0 ) {
    cout « "Hittade ’finns inte’ på index " « Found - S « endl;
  }
  else {
    cout « "Hittade inte ’finns inte’" « endl;
  }
}
```

De parametrar som `strstr()` behöver är enligt `man`-sidan (se Avsnitt A.8) en "höstack" och en "nål", där man söker nålen ur höstacken. Kör man programmet ovat får man exakt samma utskrift som tidgare.

A.8. Mera information

Alla de funktioner som räknats upp i dett kapitel som vi inte själv skrivit har egna `man`-sidor. Dessa är referenser som berättar hur funktionerna används och vilka parametrar de tar. Man kan t.ex. se på `man`-sidan för `strcat()` med kommandot:

```
% man strcat
STRCAT(3)          Linux Programmer’s Manual          STRCAT(3)

NAME
       strcat, strncat - concatenate two strings

SYNOPSIS
       #include <string.h>
```

```
       char *strcat(char *dest, const char *src);
       char *strncat(char *dest, const char *src, size_t n);

DESCRIPTION
       The strcat() function...
%
```

Alla normala funktioner har `man`-sidor under Unix. Under Linux finns en samlingssida för alla strängrelaterade funktioner på `man`-sidan `string`. Så kommandot `man string` ger en lista på alla funktioner som kan användas för att manipulera strängar, samt referenser vidare till de olika funktionernas `man`-sidor.

Appendix B. Input/output på C:s vis

Detta kapitel behandlar hur man sköter input/output i C. Denna funktionalitet kan även användas i C++ om så skulle behövas.

B.1. Introduktion

Alla exempelprogram i detta kompendium har använt sig av de former för input och output som finns i C++. Dessa skiljer sig väldigt mycket från det som traditionellt använts i C, och vi skall nu se på hur man kan hantera I/O i C. Alla former av I/O som är giltig C är även giltig C++, så följdaktligen kan man skapa ny C++-program som använder sig av C:s I/O-rutiner.

B.1.1. Buffring

Även data som läses och skrivs i C använder sig av *buffring*, vilket kan leda till oväntade resultat för olika operationer omman inte är medveten om buffringen. Läs mera om buffring i Avsnitt 8.1.

B.1.2. Headerfiler

Alla former av I/O-funktioner i C finns definierade i headerfilen `<stdio.h>` (eller `<cstdio>` enligt C++-standarden). Det räcker för det mesta med att inkludera denna fil för att använda sig av C:s I/O funktioner.

B.1.3. Konceptet med filer

Allting i C som man kan läsa och skriva till är *filer*. I C++ är alltig *streams*. Dessa skiljer sig konceptuellt inte speciellt mycket från varandra, men sättet som de används

är helt olika. Vissa funktioner använder sig av namnet `stream` för parametrar, men det är dock alltså inte frågan om C++-streams.

Man kan skriva data till olika filer i C. Dessa är:

- *skärmen* som betecknas *standard output*, och motsvaras av `cout` i C++. All output går till denna fil om inte annat anges.
- *errorfilen* som betecknas *standard error* och motsvaras av `cerr` i C++. Denna används för felmeddelanden.
- *filer* som är normala fysiska filer på disken.

I C kan man läsa data från:

- *tangentbordet* som betecknas *standard input*.
- *filer* som är normala fysiska filer på disken.

Man har gett *standard output*, *standard input* och *standard error* kortare namn somanvänds då man anger att man vill läsa eller skriva från/till dem. Dessa är `stdin`, `stdout` och *stderr*. Egentligen är dessa globala variabler som finns definierade i och med att man inkluderat `<stdio.h>`, precis som `cout`, `cin` och `cerr` i C++. Varje I/O-operation som utförs måste alltid använda sig av någon av ovanstående fördefinierade filer, eller så en fysisk fil på disken.

B.2. Utskrift till skärm

Utskrifter till skärm sköts av funktionen `printf()`. Den definieras på följande sätt:

```
#include <stdio.h>
int printf(const char * format,  ...);
```

Denna funktion är en mycket mångsidig funktion, och kan hantera väldigt många olika former av utskrifter. Ser man på funktionsdefinitionen lite noggrannare ser man att den andra parametern är `...`. I detta sammanhang innebär `...` att där kan finnas ett fritt antal extra parametrar, allt från inga till väldigt många. Första parametern är en C-sträng som anger *hur* utskriften skall formateras, d.v.s. fältbredder, datatyper o.s.v., medan eventuella variabler och konstanter definieras separat istället för `...`. Ett exempel på hur programmet `Hello` kan skrivas i C:

```
#include <stdio.h>

int main () {
  printf ("Hello world!\n");
}
```

I detta fall så använder vi endast den första parametern för `printf()`, och fyller där i den text vi vill skriva ut. Inga extra variabler eller dylikt skall skrivas ut, så inga fler parametrar behövs. Notera att vi inkluderar endast `<stdio.h>`, och inte alls `<iostream>` som vi tidigare använt i så gott som alla program.

B.2.1. Utskrift av variabler

För att skriva ut variabler till skärmen måste vi använda extra parametrar till `printf()`. Vi måste i så fall även specificera i formatsträngen *var* och *hur* vi vill att variablerna skall skrivas ut. Här används vissa platshållare inne i formateringssträngen. Dessa reserverar plats för egentliga variabler som skrivs ut istället för platshållaren. Olika typer av platshållare används för olika datatyper, men alla har ett procenttecken (`%`) som första tecken för att indikera att det är ett platshållare. De vanligaste typerna är:

- `%d` för att skriva ut ett heltal (`int`).
- `%c` för att skriva ut ett tecken (`char`).
- `%s` för att skriva ut en C-sträng.
- `%f` för att skriva ut ett flyttal (`float` eller `double`).
- `%%` för att skriva ut ett normal procenttecken.

Ett exempel på en utskrift där man skriver ut ett heltal följt av ett flyttal kan se ut på följande sätt:

```
#include <stdio.h>

int main () {
  int Antal = 10;
  float Medeltal = 13.23;
  printf ("Antalet tal: %d, medeltalet är %f\n", Antal, Medeltal);
}
```

Vi har här skrivit in strängarna `%d` och `%f` på de ställen där vi vill att ett heltal respektive ett flyttal skall skrivas ut. Annan text kan placeras hur som helst runt dessa platshållare. Eftersom vi i formateringssträngen specificerar att vi skall skriva ut två tal måste dessa även ges till `printf()`, och det görs genom att ge två extra parametrar, d.v.s. `Antal` och `Medeltal`. Kör vi programmet ovan får vi:

```
% ./COutput2
Antalet tal: 10, medeltalet är 13.230000
%
```

Genom att placera in olika platshålare för variabler och konstanter i formateringssträngen kan vi uppnå mycket komplexa utskrifter. För att t.ex. skriva ut några C-strängar kan man göra på följande sätt:

```
#include <stdio.h>

int main () {
  char * S1 = "Här är";
  char * S2 = "en sträng i";
  char * S3 = "delar";

  printf ("%s %s %d %s.\n", S1, S2, 3, S3);
}
```

Har har vi ingen egentlig "ren" text i formateringssträngen om man borträknar mellanslagen, punkten och radbytet. Istället finns där fyra platshållare för tre strängar och ett heltal. Här används `%s` för att representera en C-sträng. Notera att `%s` inte kan användas för att direkt skriva ut variabler av typen `string`, utan man måste i så fall använda metoden `c_str()`. Kör man programmet ovan får man följande utskrift:

```
% ./COutput3
```

```
Här är en sträng i 3 delar.
%
```

Man kan skriva platshållare helt fast i varandra om så skulle behövas, så ovanstående kunde även skrivas `%s%s%d%s`, förutsatt att man har lämpliga tomrum inne i strängarna istället.

B.2.2. Utskrift av flyttal

Flyttal är en aning speciella att skriva ut i C, precis som i C++. Man kan justera antalet decimaler som talet skall skrivas ut med, samt även den minimala fältbredden som skall användas. För att justera antalet decimaler kan man sätta en punkt (`.`) före `f` i `%f` och efter punkten skriva antalet decimaler som önskas. T.ex.:

```
#include <stdio.h>
#include >math.h>

int main () {
  printf ("3 decimaler: %.3f\n", M_PI );
  printf ("6 decimaler: %.6f\n", M_PI );
  printf ("0 decimaler: %.0f\n", M_PI );
}
```

Vi har här inkluderat `<math.h>` för att få tillgång till en definition av *pi* med hög precision. Vi skriver sedan ut *pi* med olika antal decimaler. Programmet ger följande utskrift:

```
% ./COutput4
3 decimaler : 3.142
6 decimaler : 3.141593
20 decimaler: 3.14159265358979311600
0 decimaler : 3
%
```

Om ingen precision ges med hjälp av en punkt och ett tal skrivs talet ut med sex decimaler.

Det finns ett stort antal olika sätt som utskrifter kan manipuleras då man använder sig av `printf()`. Mera information finns i `man`-sidan för `printf()`.

B.2.3. Utskrift teckenvis

Ibland kan det vara bra att kunna skriva ut tecken till skärmen ett i sänder. I så fall kanske det är onödigt att använda sig av något i stil med:

```
printf("%c", Tecken);
```

för att åstadkomma utskriften, eftersom `printf()` måste göra en hel del extra jobb innan tecknet `Tecken` kan skrivas ut. Det finns en funktion som kan användas för att skriva ut ett enda tecken på skärmen, nämligen:

```
#include <stdio.h>
int putchar(int c);
```

Denna funktion skriver ut tecknet `c` direkt till skärmen. Vi kan skriva ut innehållet i e C-sträng med följande funktion:

```
#include <stdio.h>

void printString (const char * String) {
  // iterera över alla tecken
  for (unsigned int Index = 0; Index < strlen(String); Index++ ) {
     putchar ( String[Index] );
  }
}

int main () {
  char * S = "Denna sträng skall skrivas ut.\n";

  // skriv ut strängen
  printString ( S );
}
```

Programmet skriver helt enkelt ut strängen `S` på skärmen.

B.3. Utskrift till filer

Att skriva till filer i C är enkelt att göra om man kan skriva till skärmen. Det enda som egentligen behöver göras är att specificera att utskriften skall gå till en annan fil än `stdout` som representerar skärmen. Därefter kan t.ex. `printf()` användas för att skriva data till filen.

B.3.1. Öppna och stänga filer

Innan man kan skriva till filer måste de öppnas. En fil öppnas i C t.ex. med funktionen `fopen()`. Den definieras på följande sätt:

```
#include <stdio.h>
FILE * fopen(const char * path, const char * mode);
```

Funktionen vill alltså ha som en första parameter ett namn på en fil. Eftersom vi vill skriva till en fil behöver filen inte existera från tidigare. Den andra parametern `mode` anger vad vi vill göra med filen. Några olika strängar kan ges:

- `r` för att öppna en fil för läsning.
- `r+` för att öppna en fil för läsning och skrivning.
- `w` öppnar en fil för skrivning. Allt eventuellt tidigare innehåll i filen (om den existerade) förstörs. Filen skapas om den inte existerar.
- `w+` för att öppna en fil för läsning och skrivning. Filen skapas om den inte existerar.
- `a` öppnar en fil för skrivning, men data skrivs till *slutet* av filen och tidigare innehåll förstörs inte. Filen skapas om den inte existerar.
- `a+` för att öppna en fil för läsning och skrivning, men data skrivs till slutet av filen.

Funktionen returnerar en pekare till en `FILE`, som är en `struct` som i C används för att representera en fil. Filerna `stdin`, `stdout` och `stderr` är av typen `FILE`. Om filen inte kunde öppnas av någon orsak kommer funktionen att returnera `0` för att indikera att öppnandet misslyckades.

En öppnad fil stängs med funktionen `fclose()`:

```
#include <stdio.h.h>
int fclose(FILE * stream);
```

Denna funktion stänger en fil som öppnats med `fopen()`. Ett exempelprogram som öppnar och stänger en fil:

```
#include <stdio.h>

int main () {
  // öppna filen för skrivning
  FILE * F = fopen ( "foo.txt", "w" );

  // kunde vi öppna den?
  if ( F ) {
// allt ok, skriv meddelande til skärmen
printf ( "Filen öppnades ok\n" );

// stäng filen igen
fclose ( F );
  }
  else {
// kunde inte öppna filen
    printf ( "Filen kunde inte öppnas!\n" );
  }
}
```

B.3.2. Skriva data till fil

När man öppnat en fil med `fopen()` kan man skriva till den med t.ex. en version av `printf()` som heter:

```
#include <stdio.h>
int fprintf(FILE * stream, const char * format,  ...);
```

Denna skiljer sig från vanliga `printf()` endast genom att den tar en extra parameter som anger vilken fil man skall skriva till. Man kan även skriva till skärmen med `fprintf()`, men då måste man explicit ange `stdout` som den fil man skriver till:

```
fprintf ( stdout, "Hello world!\n" );
```

Vi kunde fylla på programmet från föregående avsnitt med olika `fprintf()`-satser, men det är inget svårt med att skriva till filer, så det lämnas som en övning till läsaren. :-)

B.3.3. Skriva till filer teckenvis

För att skriva till filer teckenvis kan man t.ex. använda sig av en funktion som heter:

```
#include <stdio.h>
int fputc(int c, FILE * stream);
```

för att skriva tecknet `c` till filen `stream`.

B.3.4. Skriva binär data till filer

I C finns en funktion `fwrite()` som kan vara praktisk då man vill skriva ut t.ex. innehållet i en vektor eller en `struct` till en fil. Den skriver ut innehållet i binär form till filen, precis som det är i minnet. Man bör dock notera att man inte kan skriva ut t.ex. pekare eller dylikt med `fwrite()`, eftersom den då skriver ut endast pekarens adress, inte det data som pekaren pekar på. Allmänt ser `fwrite()` ut på följande sätt:

```
#include <stdio.h>
size_t fwrite(void * ptr, size_t size, size_t nmemb, FILE *
stream);
```

Parametrarna kräver en aning extra förklaringar. Den första parametern `ptr` är en pekare som pekar till det data som skall skrivas ut. Den andra är storleken på ett element som skall skrivas ut. Denna storlek kan fås med t.ex. `sizeof()`. Den tredje parametern `nmemb` (number of members) anger hur många element som skall skrivas ut. Om det är frågan om en vektor med t.ex. 10 element ges denna parameter värdet 10. Den sista parametern är filen som data skall skrivas till.

Mera information om funktionen finns i t.ex. `man`-sidan.

B.4. Inläsning från tangentbordet

Att läsa in från tangentbordet i C är inte lika enkelt som i C++. Man måste utföra noggrannare felkontroller manuellt och i övrigt göra mera arbete. Man kan antingen läsa i data *teckenvis* eller så kan man läsa in mera data på en gång med hjälp av mera avancerade funktioner. Läser man teckenvis kan man använda `getchar()` och läser man mera på en gång kan man använda `scanf()`. De definieras allmänt:

```
#include <stdio.h>
int getchar(void);
int scanf(const char * format, ...);
```

Av dessa är `getchar()` mycket lätt att använda. Den returnerar nästa tecken som skrivits på tangentbordet. Om ingenting finns väntar funktionen på att något skall skrivas och `enter` tryckas. Ifall inget kunde läsas returneras konstanten `EOF`. Denna indikerar att filslutet nåtts och inget kunde läsas, eller ett annat fel inträffat. Vi kan göra ett enkelt program som skriver ut det som skrevs på skärmen:

```
#include <stdio.h>

int main () {
  char c;

  // läs tecken ända tills vi får ett EOF (end of file)
  while ( (c = getchar ()) != EOF ) {
    putchar ( c );
  }
}
```

Programmet kan avbrytas genom att trycka `control` och `d` för att generera en end-of-file åt programmet.

B.4.1. Inläsning radvis från tangentbordet

Vill man läsa en hel rad från tangentbordet istället för enskilda tecken kan ma använda funktionen `gets()`, som allmänt ser ut på följande sätt:

```
#include <stdio.h>
char * gets(char * s);
```

Parametern `s` är pekare till en C-sträng, d.v.s. en buffer dit data kan läsas. Data läses till filen tar slut (end-of-file) eller ett radbyte (`\n`) läses. Radbytet ersätts i buffern `s` med ett termineringstecken `\0`. Denna funktion är dock ganska farlig i och med att den inte kontrollerar bufferns längd på något sätt, så om buffern är för liten utför funktionen ett fel. Undvik `gets()` om du inte vet vad du gör!

B.4.2. Formaterad inläsning

Ifall man inte vill läsa in från tangentbordet tecken- eller radvis, utan vill ha mera funktionalitet kan man läsa formaterad data med hjälp av `scanf()`. Den är definierad allmänt tidigare. Man kan tänka sig att `scanf()` är motsatsen till `printf()` och `fprintf()`. Man skriver formateringssträngen på samma sätt, dock en aning mera begränsade möjligheter, och istället för variabler ges pekare till variabler dit data kan skrivas. Så för att läsa in ett heltal följt av ett flyttal kan följande användas:

```
int Heltal;
float Flyttal;

scanf ("%d %f", &Heltal, &Flyttal);
```

Funktionen försöker nu konvertera nästa text som läses in från tangentbordet till ett heltal följt av ett mellanslag och därefter ett flyttal. Om något annat påträffas misslyckas inläsningen. Funktionen `scanf()` returnerar alltid antalet parametrar som kunde läsas in. I fallet ovan returneras två ifall båda talen kunde läsas in, och noll eller ett om något gick fel. Så vi borde egentligen skrivit:

```
int Heltal;
```

```
float Flyttal;

if (scanf ("%d %f", &Heltal,
                    &Flyttal) ) != 2 ) {
  // inläsningen misslyckades!
  ...
}
```

Alla datatyper som kan skrivas ut med `printf()` kan läsas in med `scanf()`. Man behöver dock inte specificera t.ex. precision och antal decimaler då man läser in flyttal.

Det är svårare att läsa formaterad data med C än med hjälp av streams i C++, eftersom streams tar hand om en hel del extra uppgifter, såsom att läsa in radslut, extra tomma tecken o.s.v.

B.5. Inläsning från fil

Vid det här laget är inläsning från fil mycket enkelt. Vi har redan behandlat alla nästan funktioner som behövs. Det första som måste göras om man vill läsa från en fil på disken är att filen måste öppnas för *läsning*. Se Avsnitt B.3.1. Då filen är öppnad för läsning kan man använda t.ex. följande funktioner för att läsa data från filen:

```
#include <stdio.h>
int fgetc(FILE * stream);
int fscanf(FILE * stream, const char * format, ...);
```

Dessa är båda varianter på `getchar()` respektive `scanf()`, med den skillnaden att man explicit anger en fil, istället för att anta filen `stdin`. Vi kan nu göra ett program som kopierar en fil effektivt:

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char * argv[]) {
  int c;
  FILE * Source;
  FILE * Destination;
```

```
  // kolla antalet argument
  if ( argc != 3 ) {
    // fel antal argument
    printf ("Fel antal argument!\n" );
    printf ("Användning: %s infil utfil\n", argv[0] );
    exit (EXIT_FAILURE);
  }

  // öppna fil för läsning
  if ( (Source = fopen (argv[1], "r" )) == 0 ) {
    // kunde inte öppna filen
    printf ("Kunde inte öppna filen '%s'\n", argv[1] );
    exit (EXIT_FAILURE);
  }

  // öppna fil för skrivning
  if ( (Destination = fopen (argv[2], "w" )) == 0 ) {
    // kunde inte öppna filen
    printf ("Kunde inte öppna filen '%s'\n", argv[2] );
    exit (EXIT_FAILURE);
  }

  // läs tecken ända tills vi får ett EOF (end of file)
  while ( (c = fgetc ( Source )) != EOF ) {
    fputc ( c, Destination );
  }

  // stäng alla filer
  fclose ( Source );
  fclose ( Destination );
}
```

Programmets egentliga huvudslinga använder sig av `fgetc()` och `fputc()` för att läsa och skriva ett tecken i taget. Tecknet läses som en `int` så att man kan ha en speciellt tecken `EOF` som inte blandas ihop med vanliga tecken. Se mera om funktionerna på deras respektive `man`-sidor.

B.6. I/O på låg nivå

Man kan i C ta hand om I/O på en mycket låg nivå, d.v.s. systemnära. Detta kapitel kommer inte att behandla hur dessa används. Följande funktioner kan användas:

```
#include <unistd.h>
```

```
ssize_t read(int fd, void * buffer, size_t count);
ssize_t write(int fd, void * buffer, size_t count);
```

Med hjälp av `read()` och `write()` kan man läsa och skriva till filer och andra enheter på en lägre nivå än då man använder datatypen `FILE`. Man kan t.ex. läsa och skriva olika typer av *sockets* som används för att implementera nätkommunikation.

Läs mera om desssa funktioner på respektive `man`-sidor.

Appendix C. Kodlistor

Detta appendix innehåller diverse kompletta kodlistor för klasser och program som skrivits i de olika kapitlen.

C.1. Klassen `Vector`

Denna klass användes i Kapitel 21. Definitionsfilen `Vector.h`:

```
#ifndef VECTOR_H
#define VECTOR_H

#include <iostream>
#include <math.h>

class Vector {
public:
  // konstruktorer
  Vector ();
  Vector (const Vector & V);
  Vector (const float X, const float Y, const float Z);

  // accessera individuella komponenter
  void setX (const float X) { m_X = X; };
  void setY (const float Y) { m_Y = Y; };
  void setZ (const float Z) { m_Z = Z; };
  float x () const { return m_X; };
  float y () const { return m_Y; };
  float z () const { return m_Z; };

  // get the length of the vector
  float length () const;

  // överlagrade operatorer
  Vector operator+ (const Vector & V) const;
  void operator+= (const Vector & V);
  Vector operator- (const Vector & V) const;
  Vector operator- () const;
  Vector operator* (const float S) const;
  bool operator== (const Vector & V) const;
  bool operator!= (const Vector & V) const;
  Vector & operator= (const Vector & V);

  // friend-funktioner
  friend Vector operator* (const float S, const Vector & V);
  friend ostream & operator« (ostream & Stream, const Vector & V);
```

```
  friend istream & operator» (istream & Stream, Vector & V);

private:

  // x, y och z-komponenterna
  float m_X;
  float m_Y;
  float m_Z;
};

#endif  // VECTOR_H
```

Implementationsfilen `Vector.cpp`:

```
#include "Vector.h"

// tom konstruktor
Vector::Vector () {
  // nollställ alla medlemmar
  m_X = 0;
  m_Y = 0;
  m_Z = 0;
}

// copy-konstruktor
Vector::Vector (const Vector & V) {
  // kopiera data från 'V'
  m_X = V.m_X;
  m_Y = V.m_Y;
  m_Z = V.m_Z;
}

// skapa från separata värden
Vector::Vector (const float X, const float Y, const float Z) {
  // spara värden i medlemmar
  m_X = X;
  m_Y = Y;
  m_Z = Z;
}

// beräkna längden av vektorn
float Vector::length () const {
  // använd Pythagoras
  return sqrt ( m_X * m_X + m_Y * m_Y + m_Z * m_Z );
}

// överlagrade operatorn '+'
Vector Vector::operator+ (const Vector & V) const {
  Vector Result;

  // sätt den nya vektorns värden
  Result.setX ( m_X + V.m_X );
  Result.setY ( m_Y + V.m_Y );
```

```
  Result.setZ ( m_Z + V.m_Z );

  // returnera den nya vektorn
  return Result;
}

// överlagrade '+='
void Vector::operator+= (const Vector & V) {
  m_X += V.m_X;
  m_Y += V.m_Y;
  m_Z += V.m_Z;
}

// överlagrade binära operatorn '-'
Vector Vector::operator- (const Vector & V) const {
  Vector Result;

  // sätt den nya vektorns värden
  Result.setX ( m_X - V.m_X );
  Result.setY ( m_Y - V.m_Y );
  Result.setZ ( m_Z - V.m_Z );

  // returnera den nya vektorn
  return Result;
}

// överlagrade unära operatorn '-'
Vector Vector::operator- () const {
  Vector Result;

  // sätt den nya vektorns värden
  Result.setX ( -m_X );
  Result.setY ( -m_Y );
  Result.setZ ( -m_Z );

  // returnera den nya vektorn
  return Result;
}

// överlagrade binära '*'
Vector Vector::operator* (const float S) const {
  Vector Result;

  // sätt den nya vektorns värden
  Result.setX ( m_X * S );
  Result.setY ( m_Y * S );
  Result.setZ ( m_Z * S );

  // returnera den nya vektorn
  return Result;
}

// överlagrade operatorn '=='
bool Vector::operator== (const Vector & V) const {
```

```
  // jämför V1 och denna vektor
  if ( m_X == V.m_X && m_Y == V.m_Y && m_Z == V.m_Z ) {
    // vektorerna är lika
    return true;
  }

  // de är olika
  return false;
}

// överlagrade operatorn '!='
bool Vector::operator!= (const Vector & V) const {
  // använd negerad jämförelse
  return ! ( *this == V );
}

// överlagrade '='
Vector & Vector::operator= (const Vector & V) {
  // tilldelas vi värdet av oss själva?
  if ( this == &V ) {
    // jep, gör inget i så fall
    return *this;
  }

  // utför tilldelning
  m_X = V.m_X;
  m_Y = V.m_Y;
  m_Z = V.m_Z;

  // returnera oss själva
  return *this;
}

// extern funktion för överlagrad '*'
Vector operator* (const float S, const Vector & V) {
  // använd tidigare överlagrad operator '*'
  return V * S;
}

// extern funktion för överlagrad '«'
ostream & operator« (ostream & Stream, const Vector & V) {
  // skriv ut till den stream vi fått som parameter
  Stream « V.x () « ' ' « V.y () « ' ' « V.z () « ' ';

  // returnera streamen
  return Stream;
}

// extern funktion för överlagrad '»'
istream & operator» (istream & Stream, Vector & V) {
  // skriv ut till den stream vi fått som parameter
  Stream » V.m_X »  V.m_Y » V.m_Z;

  // returnera streamen
```

```
  return Stream;
}
```

Referenser

Några referenser till läsvärd litteratur och bra hemsidor.

Böcker

The C++ Progamming Language, 3rd Edition, Bjarne Stroustrup, 1999, Addison-Wesley.

Design Patterns: Elements of Reusable Object-Orineted Software, Erich Gamma, Richard Helm, Ralph Johnson, och John Vlissides, 1995, Addison-Wesley.

Advanced Programming in the UNIX Environment, W. Richard Stevens, 1994, Addison-Wesley.

STL Tutorial and Reference Guide, David R. Musser och Atul Saini, 1996, Addison-Wesley.

Reliable Object-Oriented Software: Applying Analysis & Design, Ed Seidewitz och Mike Stark, 1995, Prentice Hall.

Hemsidor

Dinkum Standard Template Library Reference Index (http://www.dinkumware.com/htm_stl/_index.html) .

SGI STL Programmer's Guide (http://www.sgi.com/Technology/STL/) .

Other STL Web sites (SGI) (http://techpubs.sgi.com/library/manuals/3000/007-3426-003/html/other_resources.html)

.

STL Tutorial (http://www.infosys.tuwien.ac.at/Research/Component/tutorial/prwmain.htm) .

STL Newbie Guide (http://www.xraylith.wisc.edu/~khan/software/stl/STL.newbie.html) .

Index

C++, 15

minnesläckor, **151**