Enkel stilguide för C++

© *Tommy Olsson, <tao@ida.liu.se>*
Linköpings universitet och Tekniska högskola
Institutionen för datavetenskap
Programvara och system
581 83 Linköping

Innehåll

Inledning

Stil är en ofta förbisedd men mycket viktig aspekt då man skriver. Sättet att skriva en text har en direkt inverkan på läsbarhet och förståelse. Programmeringsstil, i meningen hur man skriver källkod, är också något som ofta förbises. Program måste vara läsbara och förståeliga för mänskliga läsare och inte bara för maskiner. Detta är viktigt för att skapa programvara av god kvalitet, som inte enbart uppfyller användarens krav utan också kan utvecklas inom givna tids- och kostnadsramar.

Ett program vars innebörd är klar och tydlig har en större sannolikhet att också vara korrekt och pålitligt. Då man ska modifiera ett program krävs en ingående förståelse av programkoden, vilket i hög grad stöds av klarhet. Underhåll av programvara (ofta en fråga om utveckling) är kostsamt och pågår under ett systems hela livstid. Klarhet spelar därmed en viktig roll i att hålla underhållskostnader nere.

Vad utmärker en klar, läsbar och begriplig text? Det kan vara allt från smått till stort. En del kan man relatera till hur vi skriver vanlig text, för att göra den läsbar och begriplig. Vi bygger upp text med lämpligt och korrekt ordval, delar upp texten i satser, som sammanställs i stycken, osv. Även insättning av mellanrum och tomma rader, liksom interpunktering är viktigt. Uppdelning av en text i stycken, avsnitt och kapitel kan man i program jämföra med t.ex. användning av sammansatta satser, underprogram, moduler och liknande. Sedan finns naturligtvis en hel del som är specifikt för programkod.

God läsbarhet uppnår man bl.a. genom enhetlighet, enkelhet (krångla inte till koden i onödan), uppdelning av större problem i mindre delproblem (söndra och härska) och genom att undvika underförstådda eller svårbegripliga egenskaper hos programspråket. Bra program skapas vanligtvis genom ett gediget ingenjörsarbete.

Många programvaruföretag har programmerarhandböcker med regler och rekommendationer för hur företagets programvara ska skrivas. Syftet är att programvara som produceras inom olika projekt och av olika programmerare ska vara lätt att läsa, förstå och underhålla och även vara enhetlig.

Denna stilguide är framtagen för att användas i kurser. Ett syfte är att försöka visa på god programmeringsstil (detaljer kan alltid diskuteras!) i allmänhet och speciellt för C++-programmering. Ett annat syfte är att dessa regler och rekommendationer ska kunna användas för bedömning av laborationsuppgifter och projekt.

Följande beteckningar används i denna stilguide (vad som ska vara regel resp. rekommendation kan naturligtvis också diskuteras):

- ℜ *Regler* markeras med indragna stycken med ett ℜ i vänstermarginalen. Regler är normalt tvingande, men det kan finnas enstaka tillfällen då det kan vara motiverat att frångå en regel.
- **!** *Rekommendationer* anges med indragna stycken med ett utropstecken i vänstermarginalen. Rekommendationer är lämpliga att följa, såvida det ej finns goda skäl att göra annat.

Stil kan anses vara en fråga om personlig smak. Så länge man skriver enbart för sig själv får man naturligtvis välja den stil man själv föredrar. Ska man arbeta i ett projekt eller i något annat sammanhang med flera inblandade, får man vara beredd på att gemensamma stilregler kan finnas och ska följas.

Källkodsfiler

Uppdelning av källkod

Moduluppdelning kan göras genom att programkod fördelas mellan olika källkodsfiler.

- deklarationer som utgör det externa gränssnittet för en modul skrivs i en *inkluderingsfil.*
- kod som implementerar en modul, t.ex. funktionsdefinitioner, skrivs i en *implementeringsfil.*
- **inline**-deklarerade funktioner skrivs med fördel i en *inline-deklarationsfil.*

Filnamn

Deklarationer av intresse för en användare, t.ex. funktionsdeklarationer, klassdefinitioner, etc., skrivs i *inkluderingsfiler.* Definitioner av funktionerna respektive klassers medlemsfunktioner skrivs i motsvarande *implementeringsfiler.*

Om det finns **`inline`**-deklarerade funktioner (kan *ej* skrivas i en implementeringsfil) är det lämpligt att placera dess på en separat **inline**-*definitionsfil*, vilken inkluderas sist i inkluderingsfilen ifråga.

För kod som utgör klass- eller funktionsmallar finns lite olika modeller för hur sådan kod ska göras tillgänglig för användande program. I vissa system finns namngivningsregler för filer innehållande mallar.

Syftet med följande regler är att ge filnamn en enhetlig tolkning utifrån filnamnens suffix.

ℜ Inkluderingsfiler i C++ ska ha filtypen `.h`, eller om man uttryckligen vill ange att det är en fil med C++-specifik kod, `.hh`. Inkluderingsfiler som innehåller kod som även accepteras av C-kompilatorer bör ha filtypen `.h`.

ℜ Implementeringsfiler i C++ ska ha filtypen `.cc`. Om den kompilator som används ej accepterar filtypen `.cc`, ska `.C`, `.cpp` , eller `.cxx` användas i stället.

ℜ Definitionsfiler med **inline**-deklarerade funktioner ska ha filtypen `.icc`.

Vissa avlusare kan ej hantera **`inline`**-funktioner. Om sådana funktioner skrivs i en separat **`inline`**-definitionsfil, kan man lösa detta med preprocessorn. Vid normal kompilering inkluderas `.icc`-filen i `.hh`-filen. Vid kompilering för avlusning, inkluderas i stället `.icc`-filen i `.cc`-filen varvid nyckelordet **`inline`** tas bort av preprocessorn, se följande exempel.

```
// Fil: Stack.hh
class Stack {
  …
};

#ifndef DEBUG
  #include "Stack.icc"
#endif
```

```
// Fil: Stack.cc
#include "STACK.hh"
#ifdef DEBUG
  #define inline
  #include "Stack.icc"
  #undef inline
#endif
…
```

Följande rekommendationer avser att ge vägledning för hur källkod bör delas upp mellan olika filer.

! En inkluderingsfil bör ej innehålla mer än en klassdefinition. Det är ofta enklare att underhålla program om denna rekommendation följs.

! En inkluderingsfil som innehåller en klassdefinition bör ha ett namn på formen *klassnamn*`.hh`.

! Placera maskinberoende kod i speciella filer, så att sådan kod blir lätt att lokalisera då koden ska anpassas till en annan maskintyp.

Inkluderingsfiler

ℜ Varje inkluderingsfil ska innehålla en mekanism för att förhindra upprepad inkludering av filen, en *gard*. Garder konstrueras med följande preprocessorteknik:

```
#ifndef STACK_HH
#define STACK_HH
...
#endif
```

ℜ Då följande slags definitioner används måste de inkluderas som separata inkluderingsfiler.

- Klasser som används som *basklasser* eller *medlemsvariabler*.
- Klasser som förekommer som *parametertyper* eller *returtyper* i funktionsprototyper eller medlemsfunktionsprototyper.
- Prototyper för funktioner eller medlemsfunktioner, vilka används i **inline**-deklarerade medlemsfunktioner som definieras i filen.

Antalet filer som inkluderas i en viss fil bör minimeras.

ℜ Klassdefinitioner för klasser som enbart refereras via pekare (`*`) eller referenser (`&`) ska *ej* infogas via inkluderingsfiler. Det är i dessa fall tillräckligt med en deklaration, t.ex.:

```
class A;   // pekare och referenser till A kan deklareras
```

Alternativt kan varje deklaration av en pekare eller referens till klassen ange nyckelordet **class**, t.ex.:

```
class B
{
  class A*  a_ptr;
};
```

ℜ Använd aldrig relativa filnamn i inkluderingsdirektiv (#**include**). Ange i kompileringskommandot eventuella tillägg då det gäller inkluderingssökvägen med preprocessorflaggan `-I`.

ℜ Varje implementeringsfil ska inkludera filer som innehåller följande slags deklarationer, om de används i de funktioner som definieras i filen:

- typ- och funktionsdeklarationer
- variabel- och medlemsfunktionsdeklarationer

! Inkludera aldrig andra filer i en `.icc`-fil.

Kommentarer

Att skriva bra kommentarer är svårt! Man ska kommentera lagom mycket och rätt saker. Ett viktigt syfte med kommentarer är att den som läser programkoden ska kunna sätta sig in i hur programmet fungerar, för att t.ex. rätta fel eller modifiera programmet. En grundförutsättning man ska anta är att läsaren är datorkunnig och har (grundläggande) kunskaper om programspråket. Kommentera därför inte sådant som är uppenbart av själva koden och inte heller detaljer.

Utgå däremot från att läsaren vet nästan inget om vad programmet ska göra. Tänk på att ett kodavsnitt som förefaller intuitivt för dig, då du skriver det, kanske inte är det för en annan läsare eller ens för dig själv senare. Kommentarer i program bör i hög grad beskriva *vad* som händer.

Ibland kan det också vara nödvändigt att förklara *hur* saker görs, t.ex. att ett komplicerat kodavsnitt kan vara en implementering av en känd algoritm (men att det kanske inte är uppenbart av koden och dessutom kan det vara värdefullt att få den upplysningen serverad) eller, om så inte är fallet, man i stället beskriver hur det går till.

Annat som är viktigt att beskriva är begränsningar, t.ex. att en datastruktur har en begränsad storlek.

Kommentering ska göras på ett kompakt och lätt lokaliserbart sätt. En *strategisk* kommentar skrivs före en klass eller ett funktion och beskriver klassens resp. funktionens syfte och egenskaper. En *taktisk* kommentar beskriver vad en enskild rad eller ett kortare satsavsnitt i koden gör. Om en taktisk kommentar avser en enskild rad bör den, om möjligt, placeras sist på raden.

Med en systematisk och väl genomtänkt namngivning och genom välstrukturerad kod blir behovet av taktiska kommentarer i koden mindre.

ℜ Varje källkodsfil ska dokumenteras inledningsvis, där dess namn och innehåll framgår.

ℜ Varje klass och funktion ska ha en strategisk kommentar som beskriver dess roll.

! Kommentera efter hand som olika programdelar färdigställs.

! Se till att kommentarer är korrekta. Var kortfattad men fullständig.

! Uppdatera kommentarer samtidigt som ändringar görs i koden.

Namngivning

Namngivning är centralt för läsbarhet och förståelse av program. Genom att tillämpa en systematiska namngivning underlättas identifieringen av olika namngivna storheter och förståelsen av koden ökar, samtidigt som behovet av kommentering minskar.

Använd alltid meningsfulla och rättvisande namn på variabler, konstanter, funktioner, datatyper, klasser, programlägen, etc.

Undvik förkortningar eftersom andra kan göra en helt annan uttydning av en förkortning än den man själv tycker är helt intuitiv.

- ℜ Använd meningsfulla namn.
- ℜ Använd normalt inte enbokstavsnamn. I vissa fall kan slingvariabler (t.ex. i en **`for`**-satser) undantas från denna regel, om det bara är fråga om att stega igenom ett antal heltalsvärden. Sådana skrivs av tradition ofta med enbokstavsnamn, `i`, `j`, ….
- ℜ Försök begränsa namn till en rimlig längd, men eftersträva hela ord.
- ℜ Namn på konstanter skrivs med genomgående stora bokstäver, t ex `MAX`.
- ℜ Namn på variabler och funktioner skrivs med genomgående små bokstäver, t ex `print`.
- ℜ Namn på abstrakta datatyper, poster (**`struct`**), typdefinitioner (**typedef**) och uppräkningstyper (**`enum`**) ska inledas med stor bokstav och för övrigt skrivas med små bokstäver, t ex `Stack`.
- ℜ I namn sammansatta av flera ord ska orden skrivas med understrykningstecken emellan, för att göra namnen enkla att läsa. Hur första bokstaven i respektive ord ska skrivas bestäms av namngivningsreglerna ovan, t ex `print_line`, `Tree_Node`, `MAX_SIZE`.
- ℜ Inled ej identifierare med enkelt eller dubbelt understrykningstecken, eller understrykningstecken följt av en stor bokstav, för att undvika namnkollisioner med biblioteksnamn.
- **!** Eftersom reserverade ord och namn i standardomgivningen är på engelska, ger det mest enhetligt intryck om man väljer engelska även för egna namn.

Anm. I en del äldre system kan t.ex. externa namn vara begränsade till endast 6 tecken och längre namn blir avhuggna med risk för att avhuggning ger namnkollisioner då namn är lika i de 6 första tecknen.

Kodningsstil

Regler och rekommendationer som rör kodningsstil kan även finnas under andra avsnitt. För klasser hänvisas helt till avsnittet om klasser.

Format

- **!** Skriv en deklaration per rad, t.ex. en variabeldeklaration eller en enkel sats per rad.
- **!** Indrag av rader från vänstermarginalen ska systematiskt avspegla det logiska nästlingsdjupet och ska vara 2, 3 eller 4 positioner (ej blandat).
- **!** Olika kodstycken (t ex delbearbetningar) avgränsas med en tomrad eller med en kommentar.

Tomrum

Tomrum spelar en viktig roll för läsbarheten. Grundregeln är att använda tomrum i enlighet med hur mellanrum sätts in och styckesuppdelning görs i vanlig text. Använd omsorgsfullt tomrum för att dela upp och dra uppmärksamhet till olika delar av program.

- ℜ Sätt in ett mellanrumstecken efter varje komma i en parameterlista, men ej före.
- ℜ Sätt in ett mellanrumstecken före och efter varje tvåställig operator. Operatorerna `->` och `.` ska dock *ej* ska ha något mellanrum före eller efter sig. I sammansatta uttryck, då det gäller deluttryck på den innersta nivån, kan man ibland utelämna mellanrum och t.ex. skriva ”`x = a*b + c*d - e/f`”.
- ℜ Sätt *ej* in mellanrumstecken omedelbart efter en vänsterparentes eller före en högerparentes.
- ℜ Sätt *ej* in mellanrumstecken före semikolon.
- ℜ Sätt in ett mellanrumstecken före en vänsterparentes, utom före en parameterlista.
- ℜ Sätt in en tom rad för att separera deklarationer från satser, för att separera satsavsnitt som motsvarar olika delmoment i en beräkning, för att separera funktionsdefinitioner från varandra, etc. Var systematisk med antalet tomma rader i olika situationer. Sätt inte in för många tomma rader i följd, en eller två kan ofta vara lagom.
- **!** Sätt ej in tomma rader som delar upp kod som hör intimt samman. T.ex. en tom rad före **while** i en **do-while**-sats ger ju lätt intrycket av att det i stället är fråga om en **while**-sats som påbörjas.

Satsblock

I C++ behöver satsblocket, till följd av de sammansatta satserna syntax, ofta komma till användning för att gruppera sammanhörande satser. Satsblockets patenteser är dock svårplacerade, om man vill ha en snygg och enhetlig kod. Hur parenteserna ska placeras har debatterats en hel del och det har utvecklats olika stilar, med sina för- och nackdelar. En variant är att placera blockparenteserna på egna rader och i samma kolumn direkt före och efter blocket. Parenteserna placeras med samma indrag som den styrsats de tillhör. Denna stil anses av många vara att föredra ur läsbarhetssynpunkt. Exempel:

```
if (...)
{
   ...
}
else
{
   ...
}
```

```
while (...)
{
    ...
}
```

```
do
{
    ...
}
while (...);
```

En liten variation på ovanstående är att dra in blockparenteserna motsvarande ett indragssteg och satserna i blocket ytterligare ett steg. Det har flera nackdelen, dels får koden ett mer ”oroligt” intryck och dessutom motsvarar inte indraget från vänstermarginalen längre det logiska nästlingsdjupet.

Ett alternativ till ovanstående är att flytta upp inledande blockparenteser till sist på raden innan. Detta ger något kompaktare kod och ytterligare fokusering på det väsentliga, nämligen nyckelorden i styrsatsen ifråga (den enda intressanta parentesen är egentligen den allra sista). Exempel:

```
if (...) {
    …
}
else {
    …
}
```

```
while (...) {
    …
}
```

```
do {
    …
}
while (...);
```

Man se ibland kod enligt exemplen ovan där **else** i **if**-atsen respektive **while** i **do**-**while**-satsen placerats efter spetsparentesen på raden innan. Det har den eventuella fördelen att eliminera en ”tom” rad men nackdelen att matchande nyckelord (**if**-**else** respektive **do**-**while**) inte får samma indrag, vilket ger en viss ”obalans”.

Styrsatser

! I styrsatser, (**if**, **else**, **for**, **while** och **do**-**while**) bör alltid satsdelen utgöras av ett satsblock, även om enbart en sats ingår. Detta medför enhetlighet, klarhet och ändringsbarhet.

! Styrsatser bör vara korta.

Funktioner

Program, som inte är *mycket* enkla, ska delas upp i funktioner. Varje funktion ska helst bara göra en sak. ”En sak” kan naturligtvis vara något komplext, men i sådana får man dela upp arbetet på flera funktioner, som var och en gör sin del.

ℜ Funktionsdeklarationer ska skrivas på *prototypformat*, dvs. med datatyperna för argumenten angivna, därför att detta ger kompilatorn möjlighet att göra typkontroll.

! I en funktions*deklaration* bör den inledande parentesen och första argumentet, om sådant finns, skrivas på samma rad som funktionsnamnet. Om utrymme finns bör även övriga argument och den avslutande parentesen skrivas på samma rad., t.ex.:

```
double fun(int, double);
```

I annat fall ska varje ytterligare parameter skrivas på en egen rad och med den avslutande parentesen direkt efter det sista parametern.

! I en funktionsdefinition kan funktionens returtyp skrivas på en egen rad ovanför funktionsnamnet. Detta gör funktionsnamnet lättare att se (det finns betydligt mer komplicerade fall än nedanstående, där fördelen med detta skrivsätt blir mer uppenbar, t.ex. i samband med mallklasser):

```
double
fun(int i, double d)
{
   …
}
```

! Funktioner bör vara korta.

Pekare och referenser

! Operatorerna `*` och & bör vara skrivna i direkt anslutning till typnamnen i deklarationer och definitioner, eftersom de är en del av typdefinitionen och följaktligen stå ihop med dess övriga delar. Ex:

```
Node*  next;
```

! Datatypbeskrivningar som innehåller operatorerna `*` och & bör typdefinieras, t.ex.:

```
typedef  Node*  Node_Pointer;
```

Meddelanden och ledtexter.

ℜ Var inte nedlåtande.

ℜ Försök inte vara humoristisk.

! Var informativ men kortfattad.

! Ange specifika inmatningsval i ledtexter, i förekommande fall.

! Ange i ledtexter om ett förvald värden kommer att användas om inget värde ges.

Utmatning

ℜ Rubricera all utmatning tydligt.

ℜ Presentera all information för användaren på ett enhetligt sätt.

Variabler och konstanter

Variabler och konstanter (symboliska konstanter) har mycket gemensamt. Det som egentligen skiljer är att en variabels värde kan ändras under programkörningen, vilket inte en konstants värde kan när den väl initierats.

Variabler

ℜ Variabler ska deklareras med minsta möjliga räckvidd. Använd inte globala variabler i onödan.

ℜ Varje variabel ska deklareras i en egen deklarationssats (använd ej multipeldeklaration).

ℜ Varje deklarerad variabel ska ges ett värde innan den används, om möjligt redan i deklarationen.

ℜ Använd, om möjligt, initiering i stället för tilldelning, för datamedlemmar i klassobjekt.

! Återanvänd inte variabler för olika syften, utan deklarera en variabel för varje syfte. Kompilatorn optimerar minnesanvändningen åt dig.

Konstanter

Namngivna konstanter (symboliska konstanter) fyller flera viktiga syften. En viktig aspekt är att ge namn åt värden, t.ex. `PI` i stället för `3.1415926535897932384`. Kod där sådana namn används blir klarare är om det litterala värdet (”magiskt värde”) används.

En annan fördel men symboliska konstanter är att konstanta värden som kan komma att ändras, t.ex. i samband med modifiering av program, ändras på *ett* enda ställe (i konstantdefinitionen).

Namngivning av värden kan också åstadkommas med variabler, men där kommer en annan viktig egenskap hos konstanter in, nämligen att de är just *konstanta*. Konstanters värden får ej ändras och den egenskapen kontrolleras av kompilatorn.

Några av reglerna för variabler ovan, kan även tillämpas för konstanter.

ℜ Konstanter ska definieras genom användning av **const** eller **enum**, aldrig med #**define** (det är gammal C-stil).

ℜ Undvik numeriska värden i koden. Använd symboliska konstanter i stället.

Uttryck

Det finns ett antal vanliga fallgropar i samband med beräkning av sammansatta uttryck. C++ har många operatorer och många prioritetsnivåer att hålla reda på, vilket bidrar till att det kan vara svårt att uttolka sammansatta uttryck korrekt, om inte uttrycken skrivs på ett sätt som stödjer uttolkningen.

För sammansatta uttryck där ingående operatorer har samma prioritet, gäller dessutom *associativitetsregler*. Dessa bestämmer om uttryck ska beräknas från vänster till höger eller från höger till vänster. Vänsterassociativitet gäller för de flesta prioritetsnivåerna, högerassociativitet endast på några.

! Använd inte kommaoperatorn, av läsbarhetsskäl.

! Den tilldelningsoperator som normalt bör användas är `=`. Övriga tilldelningsoperatorer (`+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `|=`, `^=`) försämrar ofta läsbarheten.

! Villkorsoperatorn `? :` bör normalt undvikas, av läsbarhetsskäl.

! Använd parenteser för att förtydliga beräkningsordningen för operatorer och deluttryck i komplicerade sammansatta uttryck.

Styrsatser

I C++ finns två styrsatser för selektion (**if** och **switch**) och tre för repetition (**for**, **while** och **do**-**while**). Vilken sats bör man använda i en viss situation, eller kvittar det? Ibland ger det sig självt, därför att en viss sats inte (enkelt) kan användas i en viss situationer. Det gäller t.ex. **switch**-satsen, vars styruttryck måste vara ett **int**-kompatibelt uttryck och dessutom måste värdena i **case**-lägena vara konstanter. Detta gör **switch**-satsen mer begränsad än **if**-satsen, men däremot passar den mycket bra i vissa fall och ger då enklare kod än motsvarande **if**-satsen.

Repetitionssatserna kan vara svårare att välja bland, även om det finns några grundregler man kan gå efter för att försöka använda ”rätt” sats för en viss typ av repetition, se rekommendationerna för respektive sats nedan. Tänk noga igenom valet av repetitionssats, ditt val kan vara väsentligt för att ge läsaren rätt signal om vad för slags repetition det är fråga om (utan att behöva lusläsa koden).

Av utrymmesskäl används den s.k. javastilen för att visa rekommenderad syntax nedan.

If-satsen

If-satsen bör i princip skrivas enligt följande syntax:

```
if (uttryck) {
   satser
}
else if (uttryck) {
   satser
}
else {
   satser
}
```

ℜ Styruttrycket i **if**-satsen ska vara ett villkorsuttryck eller en boolsk variabel eller boolsk funktion.

! Om samtliga uttryck beror av samma variabel bör en **switch**-sats användas i stället för en **if**-sats.

! **if**-satser bör ha maximal 7 grenar och ett maximalt nästlingsdjup av 3.

switch-satsen

switch-satsen bör i princip skrivas enligt följande syntax:

```
switch (uttryck) {
  case värde:
    satser
    break;

  case värde:
  case värde:
    satser
    break;

  default:
    satser
    break;
}
```

- En **switch**-sats ska alltid innehålla ett avslutande **default**-alternativ, i vilket oväntade fall hanteras.
- Koden i ett **case**-alternativ i en **switch**-sats ska avslutas med **break**, eller **return**. Då flera **case**-alternativ delar på samma kodblock används **break** endast för det sista av alternativen. Sådana fall bör kommenteras.

for-satsen

for-satsen bör i princip skrivas enligt följande syntax:

```
for (initieringssats uttryck2; uttryck3) {
  satser
}
```

- **for**-satsen ska endast användas för repetitioner där antalet varv kan bestämas då satsen börjar, dvs. då $uttryck_2$ är ett konstant uttryck under exekveringen av **for**-satsen, och då slingvariabeln stegas med ett konstant steg i varje varv. I annat fall används **while** (förtestade slingor) eller **do-while** (eftertestade slingor).

```
for (int index = 0; i < number_of_elements; ++index) {
  cout << vector[index];
}
```

while-satsen

while-satsen bör i princip skrivas enligt följande syntax:

```
while (styruttryck) {
  satser
}
```

! **while**-satsen bör endast väljas då styruttrycket på ett naturligt sätt kan förtestas. Genom detta undviker man ologiska initieringar av variabler som ingår i uttrycket (ofta är ett tecken på att **do-while** bör väljas i stället). Det kan vara aktuellt att avbryta slingan utan att utföra den någon enda gång.

```
p = list;
while (p != 0) {
  cout << p->name << endl;
  p = p->next;
}
```

do-while-satsen

do-while-satsen bör i princip skrivas enligt följande syntax:

```
do {
   satser
}
while (styruttryck);
```

! **do-while** används då satserna i slingan alltid ska utföras minst en gång. I annat fall används **while**.

```
cout << ”Ange övre gräns (heltal större än 0): ”;
do {
   cin >> upper_limit;
   if (upper_limit < 1)
      cout << ”Övre gräns ska vara större än 0, ge ett nytt värde: ”;
}
while (upper_limit < 1);
```

Övrigt rörande styrsatser

ℜ Använd aldrig **goto** för normal programstyning.

I tidskritiska tillämpningar eller för hantering av felsituationer kan **`goto`** tillåtas. Sådana fall skall ingående motiveras. **`goto`** bryter den sekventiella exekveringen på ett sätt som ofta gör koden svår att förstå. Dessutom finns begränsningar för när **`goto`** kan användas; det är t.ex. ej tillåtet att hoppa förbi en sats som initierar ett lokalt objekt som har en konstruktor.

! Använd **unsigned** för variabler som rimligtvis ej kan anta negativa värden. Variabler som representerar storlek eller längd är lämpliga att deklarera som **unsigned**. Genom detta kan en del otrevliga fel undvikas.

Det finns tillfällen då **`unsigned`** kan ge upphov till problem. Följande **`for`**-sats kommer aldrig att avslutas, till följd av moduloaritmetiken som används för heltalstyper utan förtecken. På en Sun SPARC kommer variabeln `i` att cykliskt genomlöpa `9`, `8`, …, `0`, `4294967295`, `4294967294`, …:

```
for (unsigned int i = 9; i >= 0; i--) …
```

! Använd alltid *inklusiva* undre gränser och *exklusiva* övre gränser, t.ex. `0 <= i` och `i < 10`. Genom att vara konsekvent i detta avseende kan svåra fel undvikas.

! Använd ej **continue** om ej påtaglig läsbarhetsvinst eller förenkling kan erhållas med. Det är ofta mer läsbart att använda **else**-gren i stället; **continue** används ju alltid tillsammans med en **if**-sats.

! Använd normalt **break** endast för att avsluta **case**-alternativ i **switch**-satser. **break** kan tillåtas för att gå ur en slinga, om detta innebär att användning av flaggor undviks.

! Skriv ej logiska uttryck på formen **if** (ptr)**, if** (!ptr) eller **if** (!!ptr), om ptr är en pekare. Många finner dessa former svåra att läsa. Skriv **if** (ptr != 0) eller **if** (ptr == 0).

! Eftersträva *en* **return**-sats i en funktion, på sista raden i funktionen. Om påtaglig förenkling kan erhållas genom flera **return**-satser, kan det tillåtas. Endast en **return**-sats stödjer regeln att varje konstruktion ska ha endast en utgång (och endast en ingång). För en funktion som returnerar **void** kan **return** helt utelämnas.

Funktioner

Funktionsparametrar

ℜ Använd inte ospecificerade funktionsargument (*ellips*, …);

! Undvik funktioner med många argument.

! Om en funktion lagrar en pekare till ett objekt som åtkoms via en parameter, ska parametern vara av typen pekare. I annat fall ska referensparameter användas.

! Använd konstanta referenser (**const** &) för värdeanrop, såvida ej parametertypen är av fördefinierad typ eller är en pekare. (Undvika kopiering och eventuellt aktivering av en destruktor för kopian).

Funktionsöverlagring

! Alla variationer av ett överlagrat funktionsnamn bör ha samma innebörd (semantik).

Returtyper och returvärden

ℜ Ange alltid en returtyp för en funktion, eller **void** om inget värde ska returneras.

ℜ En funktion ska aldrig returnera en referens eller pekare till en lokal variabel.

Inline-funktioner

ℜ Använd *ej* preprocessordirektivet #**define** för att generera effektivare kod; använd i stället **inline**-deklarerade funktioner.

! Använd **inline**-deklaration endast då detta verkligen är påkallat.

Temporära objekt

! Minimera antalet temporära objekt som skapas för funktionsparametrar och som returvärden.

Allmänt

! Undvik långa och invecklade funktioner. Försök att dela upp komplicerade funktioner i flera små.

Pekare och dynamisk minneshantering

Användning av pekare och dynamisk minneshantering kräver noggrannhet. En viktig aspekt är återlämning av dynamisk minne. Görs inte det korrekt uppstår s.k. minnesläckor (minne som inte längre används av programmet kan inte återanvändas, därför att det fortfarande är bokfört som upptaget), vilket kan medföra att minnet tar slut.

Pekare

ℜ Använd ej C-makrot `NULL` för test och tilldelning av pekare, använd hellre `0`. Om man vill använda `NULL` ska man själv definiera `NULL` enligt följande.

```
const int NULL = 0;
```

! Undvik om möjligt pekare till pekare, `**`. Ofta kan referens till pekare användas i stället, `*&`, t.ex. i samband med funktionsparametrar.

! Använd **typedef** för att förenkla programsyntaxen vid deklaration av länkade strukturer och funktionspekare.

Dynamisk minneshantering

ℜ Använd ej `malloc`, `calloc` eller `free`, för att tilldela och återlämna dynamiskt minne, utan **new** och **delete**. Om, av någon anledning, de förra används, blanda aldrig med **new** och **delete**!

! Skapa ej dynamiskt minne och förvänta att någon annan ska återlämna det.

! Tilldela alltid en pekare som pekar till återlämnat minne ett nytt värde (`0` om ej annat). Om pekarens deklarationsblock är på väg att lämnas, är detta naturligtvis ej nödvändigt.

Klasser

Klasser bör i princip definieras enlighet följande syntax, javastil t.h.:

```
class Derived : public Base
{
   public:
      …
   protected:
      …
   private:
      …
};
```

```
class Derived : public Base {
   public:
      …
   protected:
      …
   private:
      …
};
```

Man kan diskutera om åtkomstskyddsdeklarerarna, **public**:, **protected**: och **private**:, ska dras in ett steg, som ovan, eller ej.

ℜ **public**-, **protected**- och **private**-avdelningarna ska deklareras i den ordningen.

Genom denna deklarationsordning kommer det som är av intresse för användaren, **public**, först i klassdefinitionen. Därefter följer **protected**-delen med sådant som är av intresse om man avser att härleda från klassen. Sist **private**-delen som är av minst intresse allmänt.

ℜ Inga medlemsfunktioner ska definieras i klassdefinitionen.

Medlemsfunktioner som definieras i klassdefinitionen blir automatiskt **inline**-funktioner, vilket ibland kan vara önskvärt. Att specificera medlemsfunktioner i klassdefinitionen gör dock att klassdefinitionen blir omfattande och svårare att läsa. Dessutom strider det mot principen att dölja information. Det är därför att föredra att skriva separata **inline**-deklarerade specifikationer för medlemsfunktionerna, vilket också har vissa fördelar, t.ex. kan en **inline**-funktion då lätt göras om till en vanlig funktion.

Åtkomsträttighet

ℜ Specificera ej datamedlemmar i en klass som **public** eller **protected**.

I gränssnitt mot andra språk kan det vara nödvändigt att definiera **public**-specificerade datakomponenter.

Att specificera en datamedlem som **public** innebär att användare av klassen fritt kan operera på en sådan datamedlem. Detta strider mot den grundläggande principen i objektorienterad programmering att data ska vara inkapslade och enbart ska kunna manipuleras på ett kontrollerat sätt, dvs. via klassens medlemsfunktioner. Om **public**-data undviks kan dess interna representation ändras utan att detta påverkar användarens kod.

Användning av **protected**-variabler i en klass rekommenderas ej, eftersom sådana variabler blir synliga i härledda klasser. Namnen på sådana variabler kan då ej ändras, eftersom härledda klasser kan vara beroende av namnen. Om en härledd klass måste komma åt data i föräldraklassen, kan detta lösas genom att göra ett speciellt l **protected**-gränssnitt i basklassen med funktioner som returnerar **private**-data. Detta behöver ej påverka effektiviteten om funktionerna definieras som **inline**-funktioner.

Inline-funktioner

! Åtkomstfunktioner bör vara **inline**-funktioner.

const-medlemsfunktioner

ℜ En medlemsfunktion som ej ändrar ett objekt tillstånd ska vara **const**-deklarerad.

Medlemsfunktioner som är **const**-deklarerade är de enda funktioner som kan anropas för **const**-objekt. Förutom att **const**-deklaration ökar möjligheterna för kompliatorn att utföra kontroller, ökar den följaktligen även funktionens användbarhet. Ibland överlagrar man en viss funktion, eller operator, för att ha olika versioner för **const**- och icke-**const**-objekt.

Konstruktorer och destruktorer

ℜ En klass som använder **new** för att skapa minne som hanteras av klassen, ska normalt ha en kopieringskonstruktor och en destruktor.

ℜ Klasser som är basklasser och som har virtuella funktioner, måste ha en virtuell destruktor.

Tilldelningsoperatorer

ℜ En klass som använder **new** för att skapa minne som hanteras av klassen ska normalt definiera en tilldelningsoperator (**operator**=).

Operatoröverlagring

! Använd operatoröverlagring sparsamt och på ett enhetligt sätt. Tolkningen ska vara intuitiv.

! När två operatorer är varandras motsats (t ex == och !=) är det lämpligt att definiera båda.

Returtyper för medlemsfunktioner

ℜ En **public** medlemsfunktion ska aldrig returnera icke-konstant referens/pekare till medlemsdata.

ℜ En **public** medlemsfunktion ska aldrig returnera en icke-**const** referens eller pekare till data utanför objektet, såvida inte objektet delar data med andra objekt.

Arv

! Undvik arv för *del-av*-relationer.

! Ge härledda klasser åtkomst till medlemsdata via skyddade åtkomstfunktioner.

Undantag

ℜ Använd undantag (*exceptions*) enbart för verkligt exceptionella händelser, hantering av fel eller påtagligt ovanliga eller viktiga situationer.

ℜ Hantera varje undantag på lämplig designnivå.

Kodmallar

Dokumentation är en viktig del i framställning av programvara. Här behandlas den form av dokumentation som utgörs av kommentarer i källkoden. Källkodskommentarer kan delas in i strategiska och taktiska kommentarer:

- en *strategisk kommentar* placeras före ett kodavsnitt, t.ex. före en moduldeklaration eller en funktionsdefinition, och beskriver vad det beskrivna kodavsnittet fyller för funktion.
- en *taktisk kommentar* rör vanligtvis en enskild rad kod, eller några rader. Om kommentaren rör en enskild rad placeras den med fördel sist på raden om detta är möjligt, i annat fall före raden ifråga. En taktisk kommentar som rör flera rader placeras före raderna, som en rubrik.

Strategiska kommentarer vänder sig i första hand till användare av en modul eller en funktion, medan taktiska kommentarer vänder sig till den som ska underhålla koden. För mycket taktiska kommentarer gör lätt koden oläslig och bör därför undvikas. Taktisk kommentering kan också ofta undvikas genom den strategiska kommenteringen, väl valda namn och välskriven kod för övrigt. Endast komplicerad kod bör kommenteras taktiskt.

Nedan visas mallar för olika slags filer, inkluderingsfiler, *inline*-definitionsfiler och implementeringsfiler, samt för strategiska kommentarer för funktioner. Att standardisera sättet att kommentera kan göra det möjligt att med lämpliga verktyg automatiskt generera annan dokumentation, t.ex. manualsidor.

Inkluderingsfiler

```
/*
 * IDA Programvaruproduktion AB (u.p.a.)
 *
 * IDENTIFIERING
 *
 * Filnamn:      Stack.hh
 * Enhetsnamn:   Stack
 * Typ:          Moduldeklaration
 * Revision:     2.1
 * Skriven av:   H. Acker
 *
 *
 * BESKRIVNING
 *
 * Denna modul ...
 *
 * REVISIONSBERÄTTELSE
 *
 * Revision  Datum   Förändringar
 *
 * 1         970319  Ursprungsversion
 * 1.1       970407  ...
 * ...
 * 2.0       970821  ...
 *
 */

#ifndef STACK_HH
#define STACK_HH

/*
 * REFERERADE BIBLIOTEK OCH MODULER
 */

#include <*.hh>
#include "*.hh"

...

/*
 * VILLKORLIG INKLUDERING AV INLINE-DEFINITIONSFILEN
 *
 * Vid normal kompilering inkluderas inline-definitionsfilen
 * "Stack.icc" här. Vid kompilering för avlusning inkluderas filen
 * i stället i "Stack.cc". Detta styrs i kompileringskommandot med
 * preprocessorflaggan "-D" och "DEBUG" som argument, dvs. "-DDEBUG".
 */

#ifndef DEBUG
  #include "Stack.icc"
#endif

/*
 * SLUT PÅ FILEN Stack.hh
 */

#endif
```

Inline-definitionsfiler

```
/*
 * IDA Programvaruproduktion AB (u.p.a.)
 *
 * IDENTIFIERING
 *
 * Filnamn:     Stack.icc
 * Enhetsnamn:  Stack
 * Typ:         Inline-definitioner hörande till modul Stack
 * Revision:    2.1
 * Skriven av:  H. Acker
 *
 *
 * BESKRIVNING
 *
 * Denna definitionsfil...
 *
 * REVISIONSBERÄTTELSE
 *
 * Revision  Datum   Förändringar
 *
 * 1         970319  Ursprungsversion
 * 1.1       970407  ...
 * ...
 * 2.0       970821  ...
 *
 */

/*
 * FUNKTION is_empty()
 * ...
 */

template <class T>
inline
int
Stack<T>::
is_empty()
{
   ...
}

/*
 * FUNKTION ...
 * ...
 */

template <class T>
inline
...

/*
 * SLUT PÅ FILEN Stack.icc
 */
```

Implementeringsfiler

```
/*
 * IDA Programvaruproduktion AB (u.p.a.)
 *
 * IDENTIFIERING
 *
 * Filnamn:     Stack.cc
 * Enhetsnamn:  Stack
 * Typ:         Definitioner hörande till modul Stack, ej inline
 * Revision:    2.1
 * Skriven av:  H. Acker
 *
 *
 * BESKRIVNING
 *
 * Denna implementeringsfil...
 *
 * REVISIONSBERÄTTELSE
 *
 * Revision  Datum    Förändringar
 *
 * 1         970319   Ursprungsversion
 * 1.1       970407   ...
 * ...
 * 2.0       970821   ...
 *
 */

/*
 * REFERERADE BIBLIOTEK OCH MODULER
 */

#include ”Stack.hh”

/*
 * VILLKORLIG INKLUDERING AV INLINE-DEFINITIONSFILEN
 *
 * Vid kompilering för avlusning inkluderas inline-definitionsfilen
 * ”Stack.icc” här, samtidigt som inline-deklarationerna avlägsnas.
 * Vid normal kompilering inkluderas filen i stället i ”Stack.hh”.
 * Detta styrs i kompileringskommandot med preprocessorflaggan ”-D”
 * och ”DEBUG” som argument, dvs. ”-DDEBUG”.
 */

#ifdef DEBUG
   #define inline
   #include ”Stack.icc”
   #undef inline
#endif
```

```
/*
 * LOKALA DEFINITIONER
 */

/*
 * BESKRIVNING
 * ...
 */

static ...     // ...

...

/*
 * LOKALA FUNKTIONER (DEKLARATIONER)
 */

void f(...);   // ...

...

/*
 * KONSTRUKTOR Stack()
 * ...
 */

template <class T>
Stack<T>::
Stack()
{
   ...
}

/*
 * LOKALA FUNKTIONER (DEFINITIONER)
 */

/*
 * FUNKTION push(const T& item)
 * ...
 */

void
push(const T& item)
{
   ...
}

/*
 * SLUT PÅ FILEN Stack.cc
 */
```

Klassbeskrivningar

Nedan visas hur klasser kan dokumenteras i inkluderingsfilen.

```
/*
 * KLASS X
 *
 * BASKLASSER
 *
 * ...
 *
 * BESKRIVNING
 *
 * ...
 *
 * TILLSTÅND
 *
 * ...
 *
 * KONSTRUKTORER
 *
 * ...
 *
 * OPERATIONER
 *
 * ...
 *
 * DATAMEDLEMMAR
 *
 * ...
 *
 * REVISIONSBERÄTTELSE
 *
 * Revision  Datum   Förändringar
 *
 * 1         970319  Ursprungsversion
 * ...
 */

class X
{
   friend ...
   ...

   public:
    ...

   protected:
    ...

   private:
    ...
};
```

Funktionsbeskrivningar

I filmallarna ovan har strategiska kommentarer för funktioner markerats med t.ex.:

```
/*
 * FUNKTION funktionsnamn
 * ...
 */
```

I realiteten ska en fullständig funktionsbeskrivning finnas, vilken kan se ut enligt följande.

```
/*
 * FUNKTION f(int i, ...)
 *
 * BESKRIVNING
 *
 * Denna funktion...
 *
 * INDATA
 *
 * ...
 *
 * UTDATA
 *
 * ...
 *
 * SIDOEFFEKTER
 *
 * ... (helst inga)
 *
 * UTNYTTJAR
 *
 * Modul:     iostream
 * Modul:     ...
 * Funktion:  g
 * Funktion:  ...
 *
 * REVISIONSBERÄTTELSE
 *
 * Revision   Datum   Förändringar
 *
 * 1          970319  Ursprungsversion
 * ...
 */

void
f(int i, ...)
{
   ...
}
```