Enkel stilguide för C och C++

Tommy Olsson, tao@ida.liu.se, Institutionen för datavetenskap, Linköpings universitetet © 2006

Det här häftet innehåller grundläggande rekommendationer för hur man bör skriva program, främst då det gäller *kodningsstil*. Häftet är avsett att användas i inledande programmeringskurser, baserade på C eller C++. Många av rekommendationerna är dock allmängiltiga.

Innehåll

1 Inledning

I programmeringens barndom var effektivitet och minnesutnyttjande A och O. Det var en naturlig följd av dåtidens datorers begränsningar då det gäller minnesstorlek och beräkningskapacitet. Idag är minnesutrymme och snabbhet i många fall inget problem. I stället har underhåll och återanvändning av programvara blivit centralt och därigenom har läsbarhet kommit att bli den ofta viktigaste aspekten på programkod.

En stor del av allt programmeringsarbete innebär att ändra i program som någon annan har skrivit. Därför ska man lära sig att skriva program på ett enkelt och lättfattligt sätt. Skriv vad du menar och krångla inte till det!

Många saker går att skriva på olika sätt. Välj det som tydligast visar vad det är frågan om, t.ex. är

```
if (number_of_cars == 0)
```

det korrekta sättet att skriva testet ”om antalet bilar är noll”. Att skriva

```
if (! number_of_cars)
```

får anses som felaktigt, eftersom detta är en logisk test ”om ej antal bilar”.

Tänk alltid på följande då du skriver program:

- Skriv för människor, inte för datorer!
- Skriv vad du menar!
- Krångla inte till din kod!

Det är dock lättare sagt än gjort. För att skriva bra program krävs att du kan programspråket bra, att du kan använda det väl och undviker svårbegripliga konstruktioner, samt en hel del rutin. Läs och experimentera – kompilatorn är en bra lärare!

2 Namngivning

De bästa sättet att skriva begripliga program är att kalla saker och ting vid deras rätta namn. Namn och därmed program blir genom detta självdokumenterande och man behöver inte ha mängder av förklarande kommentarer. Hur ska man då välja namn? Följande riktlinjer ger i allmänhet bra namn.

- Namn på datatyper ska vara substantiv i singular:

```
enum Day {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY};

struct Date {
   int  year;
   int  month;
   int  day;
};
```

- Datakomponenter i strukturer (**struct**) och klasser (**class**) ska vara substantiv, se t.ex. `Date` ovan.
- Boolska variabler, konstanter och parametrar ska vara adjektiv eller predikatuttryck:

```
bool  empty;                // adjektiv
bool  list_is_empty;        // predikatuttryck
```

- Variabler, parametrar och konstanter, som ej är boolska, ska vara substantiv i singular:

```
Day   today;
Date  examination_date;
```

- Funktioner som ej returnerar något värde (**void**) ska vara verb:

```
void get_next_token(char* token);
void create(List& list);
```

- Boolska funktioner ska vara predikatsuttryck:

```
bool is_empty(List);
bool is_last_token();
```

- Funktioner som ej är boolska ska vara substantiv:

```
int length(List);
Token pop(Stack);
```

Det är naturligtvis enklast att namnge saker på sitt eget modersmål. Ett problem är att de flesta programspråk endast tillåter bokstäver ur det engelska alfabetet i namn. Hur gör man då med t.ex. svenskans å, ä och ö?

En dålig lösning är att använda a i stället för å och ä, och o i stället för ö, eftersom det lätt ger oönskade effekter. Ett program som behandlar datum kan ha variabler för år, månad och dag och kanske en funktion för skottår. I så fall skulle ”år” bli ”ar” (en ytenhet, inte en tidsenhet), ”månad” bli ”manad” och ”skottår” bli ”skottar” (en nationalitet eller något man gör med snö). En annan lösning vore att skriva ”aa”, ”ae” och ”oe” i stället för å, ä respektive ö, men det är också en dålig lösning.

Det bästa är att använda engelska. Eftersom reserverade ord och fördefinierade namn i programspråken är på engelska har detta dessutom fördelen att programmen blir språkligt enhetliga. Att hitta bra engelska namn är dock inte så lätt och man bör ha en svensk-engelsk ordlista till hands.

Använd meningsfulla och rättvisande namn

En variabel som anger antalet bilar i ett bilregister ska naturligtvis inte kallas `x` eller `kurt`, utan `antal_bilar`. Det finns undantag, t.ex. i korta och lätt överblickbara slingor använder man ofta enbokstavsnamn på styrvariabler (vanligtvis `i`, `j`, `k`, osv.):

```
for (int i = 0; i < number; ++i)
   cout << vector[i] << endl;
```

Om man har flera slingor nästlade i varandra kan det bli svårt att hålla ordning på sådana namn. Använd tydligare namn i sådana fall:

```
for (int row = 0; row < n_rows; ++row)
   for (int column = 0; column < n_columns; ++column)
      cout << matrix[row][column] << endl;
```

I vissa fall kan namn på funktionsparametrar med fördel väljas korta:

```
void exchange(int& x, int& y)
{
   const int  old_x = x;
   x = y;
   y = old_x;
}
```

I det här fallet är det även (tvärt emot vad som annars gäller) en fördel med neutrala namn som `x` och `y`. Vad variabler som det ska bytas värde på står för är inte intressant i funktionen `exchange`.

Riktigt illa är det om man använder namn som i sig är meningsfulla men som avser fel sak, t.ex. om man kallar en funktion som skriver ut *alla* bilar i ett bilregister för `print_car`, i stället för som sig bör `print_all_cars` eller `print_cars`. Ett annat exempel kan vara att en variabel som heter `number_of_cars` egentligen anger antalet bilar *minus ett*.[1]

I en del äldre C-system kan namnen på externdeklarerade variabler och funktioner vara begränsade till enbart 6 tecken. Det medför att namnen `print_all_cars`, `print_car` och `print_owner` alla huggs av till `print_`.

1. Använd kommandot `M-/` (alt. `ESC /` eller `M-X dabbrev-expand`) i Emacs för att slippa skriva långa namn. Skriv några inledande tecken i ett namn och slå sedan `M-/`. Närmast föregående namn som överensstämmer fyllas i.

Meningsfulla namn kan ibland innebära långa namn och det är inte bra med allt för långa namn. Försök i sådana fall hitta en kortare synonym.

Undvik förkortningar

En funktion som skriver ut alla bilar ska naturligtvis inte heta `blaha`, men den bör inte heller ha en kryptisk förkortning som `pac` (**p**rint **a**ll **c**ars). Kalla funktionen `print_all_cars`.

Egna påhittade förkortningar ska man vara mycket försiktig med. Sådana namn kan lätt feltolkas av andra. Använd hellre en kortare synonym är att förkorta ett långt namn.

Följ konventioner

I många programspråk har vissa konventioner utvecklats. Ska man bryta mot dem, bör man ha goda skäl för att göra det. Till exempel finns det konventioner för vad som skrivs med stora respektive små bokstäver, att man använder understrykningstecken i namn som består av flera ord, etc.

- Konstanter brukar skrivas med enbart stora bokstäver:

```
const int  MAX_NUMBER_OF_CARS = 500;
```

- Egendefinierade datatyper skriver man ofta med stor bokstav först (vilket dock avviker från konventionen för de fördefinierade datatyperna i C/C++):

```
struct List_Node {
   char*       word;
   List_Node*  next;
};

typedef List_Node*  List;
```

- Variabelnamn, parameternamn och funktionsnamn skrivs ofta med genomgående små bokstäver:

```
unsigned int  number_of_customers = 0;
bool          error = false;
```

Det förekommer att man använder suffix för att särskilja olika namn, t.ex. `_t` för att ange typ och `_p` för att ange pekare. Det råder delade meningar om sådana suffix' värde och det passar egentligen inte i C++.

3 Kommentarer

Syftet med kommentarer är att hjälpa läsaren förstå koden. Felstavade, grammatiskt felaktiga, tvetydiga eller ofullständiga kommentarer motverkar detta syfte. Om det är det värt mödan att lägga till en kommentar är det också värt mödan att se till att den är korrekt.

Källkod ska skrivas med självförklarande namn och enkla, begripliga programstrukturer ska användas. Detta reduceras behovet av kommentarer. Dåligt skriven kod är svår att förstå, underhålla och återanvända, oberoende av kommentarer.

Kommentarer bör endast användas för att förklara sådant som inte beskrivs av koden och för att framhäva sådant som är speciellt värt att uppmärksamma. Upprepa aldrig sådant som direkt kan utläsas ur koden. Där kommentarer behövs ska de vara koncisa, kompletta och lätt kunna särskiljas från koden.

Att skriva bra kommentarer är svårt. Är man ovan som programmerare är det extra svårt att skriva bra kommentarer, eftersom t.ex. kommentarer som rör implementering ska vara skrivna för programmerare som behärskar programspråket väl. Ofta ser man i nybörjarprogram kommentarer som beskriver detaljer och sådant som är uppenbart av koden, inte sällan ren omformulering av vad koden säger. Försök tänka dig att du är en kunnig programmerare som ska sätta dig in ett program som någon annan har skrivit, t.ex. för att rätta fel eller göra ändringar. Skriva kommentarer utifrån vad du själv skulle vilja veta i den situationen.

Nedan följer anvisningar till vad som bör kommenteras och hur du utformar kommentarer.

Kommentarer i filhuvuden

Varje källkodsfil bör ha ett filhuvud där det framgår vem som äger filen, vem som ansvarar för filen, samt en revisionshistorik. Ett exempel hur en kommentar i ett filhuvud kan se ut:

```
/*
 * Copyright (c) 2006, IDA Software.
 *
 * Ansvarig : H. Acker
 * Avdelning: Programvara och system
 *
 * Revisionshistorik:
 *   06-09-12 C. Racker
 *     Lade till funktionen delete.
 *     Rättade ett fel i funktionen find, som gav segmentation fault.
 *   06-09-11 H. Acker
 *     Originalversion.
 */
```

I C/C++ hittar man främst två slags källkodsfiler, inkluderingsfiler (”headers”) och implementeringsfiler. Inkluderingsfiler riktar sig till användare av koden, medan innehållet i implementeringsfiler främst är av intresse för de som ska underhålla koden. Detta bestämmer vad kommentarer i respektive typ av fil ska innehålla, utöver vad som angivits ovan. Hur kommentarer i olika slags filer kan utformas beskrivs nedan.

Kommentarer i inkluderingsfiler

En kommentar i en inkluderingsfil är till för att hjälpa en användare att förstå hur enheten ifråga ska *användas*. Genom att läsa kommentarer och kod, ska användaren få reda på allt som behövs för att kunna använda koden. Denna typ av kommentarer bör omfatta:

- enhetsnamnet eller liknande.
- en kortfattad förklaring av syftet (*vad* som görs, inte hur eller varför).
- en beskrivning av hur funktioner kan användas och vilka effekter de har.
- eventuella begränsningar, globala effekter, för- och eftervillkor, prestanda.
- fel som kan inträffa och hur dessa kan detekteras.

Ta *inte* med sådan information om användning som framgår av koden. Räkna inte upp funktionernas parametrar. Exempel:

```
/*
 * graphlayout
 *
 * Syfte:
 *   Den här enheten beräknar positionsinformation för noder och bågar
 *   i en riktad graf. Detta grundas på en algoritm som minimerar
 *   antalet korsande bågar och framhäver den primära riktningen för
 *   bågarna i grafen.
 *
 * Effekter:
 *  - Förväntat användningssätt är:
 *    1. Anropa define för varje nod som ska ingå i grafen.
 *    2. Anropa connect för varje par av noder som ska vara förbundna
 *       med en båge.
 *    3. Anropa layout för att tilldela varje nod i grafen en position.
 *    4. Anropa position_of för att erhålla en nods position.
 *  - Operationen layout kan anropas upprepade gånger och beräknar då
 *    om positionerna för samtliga noder.
 *  - Då en nod definierats, kommer den att vara definierad till dess
 *    clear anropas för att radera hela grafen.
 */
```

Hur datatyper, funktionsdeklarationer, etc., som utgör koden i inkluderingsfilen kommenteras beskrivs nedan.

Kommentarer i implementeringsfiler

En kommentar i en implementeringsfil är till för att hjälpa en underhållare att förstå implementeringen, även val som gjort mellan olika implementeringsalternativ. Dokumentera alla beslut som tagits vid implementeringen, så att en underhållare kan undvika att göra om samma misstag som du gjort. En viktig typ av information är en beskrivning av varför en viss lösning som övervägts inte skulle fungera. Portabilitetsaspekter hör också hemma här. Den här typen av kommentarer bör omfatta:

- enhetens namn eller liknande.
- förklaringar av *hur* och *varför* enheten utför sin uppgift, inte vad den gör.
- korta sammanfattningar av komplexa algoritmer.
- anledningar till påtagliga eller kontroversiella implementeringsbeslut.
- förkastade implementeringsalternativ, tillsammans med orsaken till detta.
- förutsedda framtida förändringar.

Ta inte med sådan information som framgår av koden. Exempel på denna typ av kommentar:

```
/*
 * Graphlayout
 *
 * Implementeringsnoteringar:
 *   Denna enhet använder en heuristisk algoritm för att minimera
 *   antalet korsande bågar. Den uppnår inte alltid det verkliga
 *   minimum som kan räknas ut teoretiskt. Den uppnår dock ett nästan
 *   perfekt resultat på förhållandevis kort tid. För detaljer om
 *   algoritmen, se …
 *
 * Portabilitetsaspekter:
 *   - Standardbiblioteket Math används för beräkning av koordinater.
 *   - 32-bitars heltal krävs.
 *   - Inga operativsystemspecifika rutiner anropas.
 *
 * Förutsedda ändringar:
 *   - Coordinate_Type skulle kunna ändras från heltal till flyttal
 *     med liten arbetsinsats. Åtgärder har vidtagits för att ej vara
 *     beroende av heltalsaritmetiska egenskaper.
 */
```

Hur funktionsdefinitioner, etc., som utgör koden i inkluderingsfilen kommenteras beskrivs nedan.

Kommentarer till funktioner

En funktion kan delas upp i en funktionsdeklaration (prototyp; specifikation) och en funktionsdefinition (kropp). Funktionsdeklarationen riktar sig till användare av funktionen, medan definitionen är av intresse för de som ska underhålla koden.

Kommentarer till funktionsdeklarationer

En kommentar till en funktionsdeklaration ska beskriva syftet med funktionen och vad funktionen gör, se nedan.

Om det är helt uppenbart vad en funktion gör och om funktionsdeklarationen är självförklarande, behöver man inte kommentera enligt ovan, utan kan t.ex. låta en övergripande kommentar vara nog.

```
/*
 * define
 *
 * Syfte:
 *   Den här funktionen definierar en nod i den aktuella grafen.
 * Undantag:
 *   Node_Already_Defined
 */
void define(const Node& new_node);

/*
 * layout
 *
 * Syfte:
 *   Den här funktionen tilldelar de definierade noderna koordinat-
 *   positioner.
 * Undantag:
 *   Inga.
 */
void layout();


/*
 * position_of
 *
 * Syfte:
 *   Den här funktionen returnerar positionen för den angivna noden.
 *   Om ingen position ännu tilldelats noden returneras den förvalda
 *   koordinatpositionen (0,0).
 * Undantag:
 *   Node_Not_Defined.
 */
Position position_of(const Node& node);
```

Kommentarer till funktionsdefinitioner

En funktionsdefinition ska ha kommentarer som riktar sig till underhållare av koden.

```
/*
 * define
 *
 * Implementeringsnoteringar:
 *   Den här funktionen lagrar en nod i den ordinära Graf-datastruk-
 *   turen, inte i Fast_Graph-datastrukturen, därför att …
 */
void define(const Node& new_node)
{
   …
}

/*
 * layout
 *
 * Implementeringsnoteringar:
 *   Den här funktionen kopierar Graf-datastrukturen (optimerad för
 *   snabb slumpmässig åtkomst) till Fast_Graph-datastrukturen
 *   (optimerad för snabb sekventiell iteration), utför layout och
 *   kopierar sedan tillbaka till Graf-datastrukturen. Denna teknik
 *   infördes som en optimering då algoritmen befanns för långsam
 *   (snabbheten förbättrades genom detta med en faktor 10).
 */
void layout()
{
   …
}
```

```
/*
 * position_of
 */
Position position_of(const Node& node)
{
   …
}
```

Om en funktion är enkel och koden självförklarande behöver man inte ha någon kommentar av ovanstående slag.

Kommentarer till data

Kommentarer till data ska förklara syfte med, struktur hos och semantik för datastrukturer. Att förstå hur data lagras och hur olika data hänger ihop kan vara ett första steg i att förstå övriga detaljer i ett program. Riktlinjer för hur du bör kommentera data:

- kommentera datatyper, variabler och konstanter, såvida inte deras namn är självförklarande.
- förklara alltid hur komplicerade, pekarbaserade datastrukturer byggs upp (datatyperna enbart säger ofta inte allt om struktur i sådana fall).
- beskriv eventuella samband mellan olika dataelement.

Exempel på kommentering av data:

```
/*
 * Dessa datastrukturer används för att lagra grafen under layout-
 * processen. Den övergripande organisationen är en sorterad lista
 * av ranker, var och en innehållande en sorterad lista av noder,
 * var och en innehållande en lista av inkommande bågar och en lista
 * av utgående bågar.
 *
 * Rank- och nodlistorna är dubbellänkade för att tillåta att listan
 * genomlöps i båda riktningarna under sorteringspassen. Båglistorna
 * behöver ej vara dubbellänkade eftersom bågordningen är oväsentlig.
 *
 * Noderna och bågarna är dubbellänkade till varandra för att tillåta
 * effektiv uppsökning av alla bågar till/från en nod, liksom effektiv
 * uppsökning av en båges start/slutnod.
 */

typedef struct Arc*   Arc_Pointer;

typedef struct Node*  Node_Pointer;

struct Node {
   Node_Pointer  id;          // unik nodidentitet, anges av användaren
   Arc_Pointer   arc_in;
   Arc_Pointer   arc_out;
   Node_Pointer  next;
   Node_Pointer  previous;
};

struct Arc {
   Arc_ID        id;          // unik bågidentitet, anges av användaren
   Node_Pointer  source;
   Node_Pointer  target;
   Arc_Pointer   next;
};
```

```
typedef struct Rank*  Rank_Pointer;

struct Rank {
   Level_ID      number;      // beräknat ordningstal för ranken
   Node_Pointer  first_node;
   Node_Pointer  last_node;
   Rank_Pointer  next;
   Rank_Pointer  previous;
};

Rank_Pointer  first_rank;
Rank_Pointer  last_rank;
```

Kommentarer till satser

Kommentarer till satser bör endast förekomma för att dokumentera kod som ej är portabel, är beroende av miljön eller på något sätt ”knepig”. Den här typen av kommentarer ska tydligt synas i koden.

Riktlinjer för kodkommentarer:

- använd minimalt kommentarer bland satser.
- använd kommentarer endast för att förklara kodavsnitt som inte är uppenbara.
- kommentera sådant som normalt borde förekomma i ett visst sammanhang men som av någon anledning medvetet utelämnats.
- använd inte kommentarer som bara är en omskrivning av kod.
- använd inte kommentarer för att beskriva kod på annan plats, t.ex. en funktion som anropas av aktuell kod.

4 Användning av variabler och konstanter

Använd symboliska konstanter och namngivna datatyper

Genom symboliska konstanter och namngivna datatyper blir program både lättare att läsa och enklare att ändra i, eftersom man ofta bara behöver ändra på ett ställe. Dåligt:

```
char  my_car[21];
```

Bättre, i alla fall om man använder längden på bilmärken på flera ställen:

```
const int  CAR_BRAND_LENGTH = 20;
char  my_car[CAR_BRAND_LENGTH + 1];
```

Bäst, i alla fall om man har fler variabler som ska innehålla bilmärken:

```
const int     CAR_BRAND_LENGTH = 20;
typedef char  Car_Brand[CAR_BRAND_LENGTH + 1];
Car_Brand     my_car;
```

Antag att vi har ett fält med etthundra bilmärken. Visst blir det tydligare med

```
const int  MAX_NUMBER_OF_CARS = 100;
Car_Brand  ten_cars[MAX_NUMBER_OF_CARS];
```

än med

```
char  cars[CAR_BRAND_LENGTH + 1][100];
```

och allra sämst är följande:

```
char  cars[21][100];
```

Vad står de ”magiska talen” 21 och 100 för?

Använd inte globala variabler i onödan

Använd inte globala variabler i onödan. En variabel ska definieras ”så nära där den används som möjligt” och det inte ska gå att komma åt den på andra ställen i programmet.

- använd parametrar och returvärden för att överföra värden till och från funktioner.
- en variabel som endast behövs i en funktion ska deklareras i funktionen.
- en variabel som ska behålla sitt värde mellan funktionsanrop men som inte används utanför funktionen kan deklareras som **static**.

Det här betyder inte att du aldrig får använda globala variabler.

Återanvänd inte variabler

Om du i en funktion först behöver arbeta med antal bilar och sen med antal bilmärken, ska du definiera två variabler kallade `number_of_cars` och `number_of_brands`). Du vinner inget på att bara definiera en variabel (kallad t. ex. `number`) och använda den för bägge ändamålen. Såna saker (att inte använda mer minne än som behövs) fixar kompilatorn åt dig.

Vissa hjälpvariabler, t.ex. slingvariabler som `i`, `j`, `k` osv., kan man dock ofta med fördel återanvända. C++-standarden tillåter t.ex. att variabler definieras i **for**-satsens initieringssats.

5 Val av selektionssatser

Valet mellan **if** och **switch** är vanligtvis inget problem. En **if**-sats styrs av ett villkorsuttryck, som kan formuleras fritt. En **switch**-sats kan bara användas när styrningen grundas på en **int**-kompatibel variabels värde. Lägena (**case**) i en **switch**-sats kan bara vara literaler av typ **int** eller **char** (eller värden ur någon **enum**-typ).

I vissa fall kan man ha en funktion som returnerar ett diskret värde, som används för att styra **switch**-satsen. I sådana fall definierar man också ibland en **enum**-typ med bra namn på uppräkningsvärdena för att användas i **case**-lägena.

6 Val av repetitionssatser

I C/C++ finns tre olika satser för repetition, **for**, **while** och **do-while**. Vilken man väljer är ibland en smaksak, men det finns i många fall ett naturligt val, beroende på vilken typ av repetition det är fråga om: *räknarstyrd* eller *villkorsstyrd*. För villkorsstyrda repetitioner kan man också hitta två former: *förtestad* och *eftertestad*.

Slingor ska vara korta. Det är svårt att få överblick av en slinga som sträcker sig över flera sidor programkod. Dela upp innehållet i komplicerade slingor i funktioner.

Räknarstyrd repetition

Räknarstyrd repetition kallas en repetition som stegar igenom en följd av (diskreta) värden, t.ex. heltalen 1 till 100 eller tecknen ’A’ till ’Z’.

Antalet varv i en räknarstyrd slinga är känt när man går in i slingan, genom ett startvärde (t.ex. 1) och ett slutvärde (t.ex. 100). Dessa värden kan bestämmas av uttryck som beräknas då slingan initieras, men under själva repetitionen ska inte slutvärdet ändras. För att styra repetitionen används en styrvariabel, som stegas från startvärdet till slutvärdet med steget 1 (eller motsvarande).

I många programspråk är **for**-satsen konstruerad för enbart räknarstyrd repetition med steget 1 (eller motsvarande). Så är inte fallet med **for**-satsen i C/C++, utan **for**-satsen i princip kan användas för godtyckliga repetioner. En god regel är dock att använda **for**-satsen för främst räknarstyrda repetitioner:

```
// Skriv ut talen 1 till last_value, decimalt och oktalt
for (int i = 1; i <= last_value; ++i)
   cout << setw(8) << i << setw(8) << oct << i << endl;
```

I C/C++ går fältindex alltid från 0 till antalet element i fältet *minus ett*. Slingor som ska stega igenom elementen i ett fält brukar därför alltid skrivas:

```
// Skriv ut de n värdena i fältet vector
for (int i = 0; i < n; ++i)
   cout << ”vector[” << i << ”] = ” << vector[i] << endl;
```

for-satsen kan även med fördel användas för att stega igenom länkade listor, t.ex.:

```
for (list_node* p = min_lista; p != 0; p = p->next)
   cout << p->data;
```

Det går bra eftersom man konstruerar styrningen av **for**-satsen själv. Detta är inte möjligt i många andra språk (t.ex. Ada och Pascal) där man för detta ändamål endast kan använda **while**-satsen (eller motsvarande). Standardbibliotekets iteratorer lämpar sig också väl för att använda i **for**-satser i många fall.

Villkorsstyrd repetition

Villkorsstyrd repetition används vanligtvis då man inte på förhand vet hur många gånger en repetition ska utföras, utan det bestäms under själva repetitionen. Villkorsstyrda repetitioner kan delas upp i två typer, *förtestade* och *eftertestade*.

Förtestad villkorsstyrd repetition

I en förtestad repetition kan det tänkas att saterna i slingan inte ska utföras någon enda gång, och testet om repetitionen ska fortsätta görs då allra först. Ett typiska exempel där denna typ av repetition används är vid läsning av filer. Filer kan dels vara tomma och dels vet vi ofta inte hur mycket data som finns i en fil, utan får läsa till filen tar slut. Exempel:

```
ifstream indatafil(”data.txt”);
while (indatafil >> x)
   sum = sum + x;
```

Ett annat typiskt fall där förtestade slingor används är stegning genom länkade listor, som kan vara tomma och som vi ofta inte lagrar längden på:

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

Eftertestad villkorsstyrd repetition

I en eftertestad repetition ska satserna i slingan utföras minst en gång och testet om fler varv ska utföras görs då sist i slingan. Sådana repetitioner kan typiskt förekomma i samband med interaktiv inmatning:

```
do {
   cout << ”Skriv ett heltalsvärde större än eller lika med 0: ”;
   cin >> value;
   if (value < 0)
      cout << ”Felaktigt värde, ” << value << ”, försök igen!” << endl;
}
while (value < 0);
```

Hopp i samband med repetitionssatser

Försök att undvika **break**-satsen i repetitionssatser. I vissa situationer är den dock användbar och ger tydligare kod än en annan lösning.

Du kommer ytterst sällan, kanske aldrig, att behöva använda **continue**-satsen. Den är i de allra flesta fall helt onödig och rör bara till koden. En **if**-sats är ofta den alternativa, rättframma lösningen.

7 Funktionsuppdelning

Dela upp program i små, enkla funktioner. Det är ett utmärkt sätt att erhålla hanterbara program och också att begränsa olika variablers räckvidd.

Varje funktion bör göra *en* sak. Denna ”enda sak” kan dock vara komplicerad och kräva många rader kod. All kod behöver inte ligga i funktionen; inför hjälpfunktioner!

Det ska helst gå att beskriva vad en funktion gör i en mening, t.ex. ”denna funktion sorterar heltal i ett fält i stigande ordning”. Om beskrivningen blir lång eller komplicerad har antagligen för mycket, som kanske egentligen inte hör dit, lagts in i funktionen. Dela i så fall upp funktionen i dess beståndsdelar.

Undvik att upprepa kod. Dels blir det onödigt mycket att läsa (och skriva), och dels måste du komma ihåg att ändra på alla ställen om du ska ändra något i koden. Samla kod som upprepas på flera ställen i ett program i en funktion, som kan anropas från de ställen där koden ska utföras.

8 Strukturerad programmering

Använd strukturerad programmering. Detta betyder att du ska använda de konstruktioner för repetition och val (**for**, **while**, **do**-**while**, **if**, **switch**) som finns i språket, och undvika hopp med **goto**, **break** och **continue**. Dessa hopp är inte förbjudna men du ska ha goda skäl för att använda dem, speciellt gäller detta **goto** och **continue**.

En del programmerare tycker att man ska vara försiktig med **return**-satsen, eftersom den fungerar som ett hopp ut ur en funktion. Varje **return**-sats blir en utgång ur funktionen och ju fler utgångar, desto svårare att överblicka hur en funktion beter sig. Varje funktion borde alltså endast returnera en gång, allra sist. Det sätt man kan behöva koda för att åstadkomma det kan ibland ge invecklad kod.

9 Formatering av kod

Hur många steg man ska göra indrag från vänstermarginalen? Hur ska man använda mellanrum och tomma rader? Ska blockparenteserna { och } stå på egna rader eller inte?

Det finns olika stilar som utvecklats för olika språk, alla med sina för- och nackdelar. Syftet med dessa är att göra program läsbara, genom att framhäva programstrukturen och genom att göra enskilda rader lätta att läsa. Programstruktur kan du framhäva med tomma rader mellan kodavsnitt och med indrag från vänstermarginalen. Lättlästa rader får du genom att sätta in mellanrum på ett genomtänkt sätt.

En del vägledning kan du få genom att tänka på hur vanligt text skrivs, med styckesuppdelning, användning av mellanrum, etc.

Om du kan välja stil själv:

- Välj en etablerad stil som passar dig och håll dig till den stilen i hela programmet.
- Var konsekvent!
- Om du ändrar i ett program som någon annan skrivit, följ den stil som använts.

Det finns program som formaterar kod, åtminstone så att alla rader får korrekt indrag. Inget program kan dock göra ett lika bra jobb som du själv!

Indrag från vänstermarginalen använder du för att framhäva den övergripande logiska strukturen i program. Många anser att tre positioner är ett optimalt indrag och att man bör hålla sig mellan 2-4 steg.

Var noggrann! Det är förödande för läsbarheten om inte programrader har korrekt indrag, i förhållande till varandra och till det logiska nästlingsdjupet.[1]

1. Använd TAB (M-X indent-line) i Emacs för att dra in varje ny rad, dels för att hela tiden ha korrekt indrag (vilket är värdefullt), dels för att kunna upptäcka vissa syntaxfel som leder till fel indrag. Du kan också markera ett område och ge kommandot M-X indent-region.

Mellanrumstecken använder du för att öka läsbarheten på enskilda rader. Ett exempel på dålig kod:

```
if(x*y>1&&(z+2*w<0||z+2*w>10))
```

Denna oläsliga gröt blir betydligt mer lättläst om du använder mellanrum för att separera namn och operatorer:

```
if (x * y > 1 && (z + 2 * w < 0 || z + 2 * w > 10))
```

Riktlinjer för användning av mellanrumstecken:

- alltid efter komma
- aldrig före semikolon
- ej före eller efter `[` och `]` (fältindexering)
- före men ej efter vänsterparentes, dock ej före vänsterparentesen i funktionshuvuden/anrop
- efter men ej före högerparentes, dock ej efter högerparentes som följs av semikolon.
- före och efter tvåställiga (binära) operatorer, utom då det gäller `->`, `.`, och `::`
- ej mellan en enställig (unär) operator och dess operand, utom då det gäller operatorn `!`.

Tomma rader använder du för att dela upp längre kodavsnitt i delmoment eller beståndsdelar, t.ex.:

- mellan funktioner som är skrivna på samma fil, för att tydligt visa var de börjar och slutar.
- mellan deklarationer och satser i ett block, för att framhäva deklarationerna.
- mellan olika delar en funktionskropp, för att t.ex. påvisa olika delmoment som utförs.

Var systematisk med antalet tomma rader i olika sammanhang. Överdriv inte antalet. Inuti funktionskroppar, blocksatser, och liknande finns det sällan anledning att ha mer än *en* tom rad då man vill göra en uppdelning. Se till att inga omotiverade tomma rader förekommer; det ger fel information och förvirrar läsaren.

Långa rader delar du upp genom att välja så vettiga brytpunkter som möjligt:

- i långa uttryck, t.ex. i en **if**-sats, gör du radbrytning mellan deluttryck; om brytningen bör göras före eller efter en operator kan bero på typen av uttryck. Dra in så att deluttrycken på de olika raderna börjar i samma position.
- långa parameterlistor i funktionsdeklarationer delar du lämpligen upp genom att skriva *en* parameter per rad (den första på samma rad som funktionsnamnet) och bryta efter varje komma. Gör indrag av raderna så att parametertypernas namn börjar med jämn vänstermarginal.
- bryt efter komma i långa argumentlistor i funktionsanrop. Dra in så att argumenten får samma vänstermarginal.

Blockparenteserna, `{` och `}`, är ett besvärligt kapitel. Var ska de placeras? Hur ska de dras in? Ska man använda dem även om blocket endast består av en sats?

Fyra exempel på stilvariationer då det gäller blockparentesers placering i en **if**-sats visas nedan. Det man egentligen vill framhäva är nyckelorden **if** och **else**, eftersom de visar vilken typ av sats det är och markerar satsens övergripande struktur. Den enda intressanta parentesen är egentligen den allra sista, som avslutar **else**-grenen, övriga parenteser är mest till besvär. En variant att skriva **if**-satsen är:

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

Den framhäver onekligen nyckelorden. Däremot försvinner till viss del den sista, avslutande parentesen genom indraget i förhållande till **if** och **else**. En påtaglig negativ effekt är att satserna i blocken får *dubbelt indrag*, fast det logiska nästlingsdjupet bara ökar med *ett* steg. Ett försök att lösa det vore att ej dra in satserna i blocken i förhållande till parenteserna, men det är ingen bra lösning, eftersom man då får olika regler i olika sammanhang där blocksatsen används. Följande kan nog betraktas som bättre och är den stil som vid tester har visat sig föredras de flesta (antagligen pga dess enkelhet, tydlighet och symmetri):

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

Parenteserna framhävs förvisso något mer, men här får vi korrekt logiskt indrag av satserna inuti blocken och den avslutande parentesen i **else**-grenen hamnar ”rätt”. Båda alternativen ovan kan ibland tyckas ge väl mycket tomrum, genom att blockparenteserna skrivs på egna rader. En variant som eliminerar det är följande:

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

Detta en iögonfallande osymmetri; det viktiga nyckelorder **else** hamnar med annat indrag än dess matchande **if** och döljs också i viss mån genom detta av parentesen som står före. En kompromiss (som f.ö. blivit standardsyntax i Java) är:

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

Här framhävs nyckelorden ordentligt och indrag blir korrekt, samtidigt som ”oönskade” parenteser döljs så långt praktiskt möjligt.

Skriv normalt endast en sak per rad, t.ex. en enkel sats eller en deklaration. Det är ofta svårt att läsa multipla deklarationer och rader med flera satser, t.ex.:

```
double values[100], sum[10], mean[100], value;
int number = 0, x;
```

Motsvarande enkla deklarationer gör det lättare att se vilka variabler som deklareras och deras typ:

```
double  values[100];
double  sum[10];
double  mean[100];
int     number = 0;
int     x;
```

Att skriva variabelnamnen med jämn vänstermarginal, som ovan, är också ett sätt att öka läsbarheten.

10 Modularisering och abstraktion

De här avsnittet handlar mer om programmeringsmetodik än om kodningsstil men är också av stort intresse att ta upp i sammanhanget.

Moduluppdelning

Att modularisera innebär att dela upp en stor uppgift i flera mindre delar. En fördel med modularisering är att det är lättare att lösa många små uppgifter var för sig än att lösa en stor uppgift. Om man gör uppdelningen på ett bra sätt, kan man ofta lösa varje del utan att behöva tänka på de andra delarna.

En annan fördel med uppdelning i mindre delar är att man kan testa varje del för sig. Då varje del fungerar som den ska, kan man slå ihop alla delarna till ett (förhoppningsvis) fungerande program.

Ytterligare en fördel med modularisering är att man kan återanvända en modul, t.ex. genom att använda den till flera olika saker i samma program eller att i framtiden använda den i andra program.

Varje modul ska vara starkt sammanhållen och kopplingen mellan moduler ska vara svag. Att en modul är starkt sammanhållen betyder att den gör *en* sak och gör denna fullständigt. Denna sak kan vara komplicerad och bestå av delmoment, t.ex. kan en moduls gränssnitt bestå av flera funktioner, som var och en utför en viss del. Att en modul är svagt kopplad innebär att den ej är beroende av hur andra moduler är implementerade.[1]

En modul kan typiskt utgöras av en uppsättning relaterade funktioner, eller en eller flera datatyper med tillhörande operationer (funktioner). Datatyper och funktionsprototyper placeras i en inkluderingsfil, medan funktionsdefinitionerna, och annat som inte en användare behöver känna till, placeras på en tillhörande inplementeringsfil.

Bygg abstraktioner

Abstraktion avser, i detta sammanhang, att dölja detaljer. Det kan gälla både data och bearbetningar. En funktion, t.ex., döljer alla detaljer i funktionskroppen. I ett anrop ser vi bara funktionens namn och de argument som motsvarar funktionens parametrar.

I vissa fall kan vi dölja data men ändå operera på data med hjälp av tillhörande operationer (funktioner). Datatyper där vi lyckats dölja och/eller förhindra åtkomst till implementeringen kallas ibland för abstrakta datatyper. En abstrakt datatyp kan ses som ett skal. Innanför skalet finns alla detaljer, t. ex. om data lagras och hur de funktioner som opererar på objekt av datatypen ifråga är skrivna. Utanför skalet varken vill man eller får man bry sig om dessa detaljer. Klasskonstruktionen i C++ gör detta möjligt.

En stack är en rak datastruktur, i vilken man kan lägga till (”push”) och ta bort (”pop”) element i dess ena ända (”sist in, först ut”). En stack kan implementeras med antingen ett fält eller med en länkad lista. Detta kan man i C++ dölja genom att kapsla implementeringen i en klass’ privata del. Den lösning vi ger nedan är mer ”C-ig”, där vi använder pekare till post (**struct**) för att dölja detaljerna.

Definitionen av en datatyp `Stack` med tillhörande operationer kan se ut enligt följande (finns på en inkluderingsfil `Stack.h`):

```
#ifndef STACK_H
#define STACK_H

typedef struct Stack_Implementation* Stack;

Stack  create();                  // anropas för att initiera en ny stack
void   destroy(Stack& s);         // anropas för att ta bort en stack
void   push(double e, Stack s);   // lägger ett värde på stacken
double top(Stack s);              // returnerar översta elementet
double pop();                     // tar bort det översta elementet
bool   empty(Stack s);            // kontrollerar om stacken är tom

#endif
```

1. De engelska begreppen för sammanhållning och koppling är *cohesion* resp. *coupling*.

En fältimplementation kan se ut enligt följande (och finns på en implementeringfil `Stack.cc`):

```
#include ”Stack.h”

const int  STACKSIZE = 100;
const int  EMPTY_STACK = -1;

struct Stack_Implementation {
   double  element[STACKSIZE];
   int     top;
};

Stack create() {
   Stack  s = new Stack_Implementation;
   s->top = EMPTY_STACK;
   return s;
}

// Övriga stackoperationer definieras här...
```

Om stacken i stället implementerats med en länkad lista kan se ut enligt följande:

```
#include ”Stack.h”

#define EMPTY_STACK 0

struct Stack_Node {
   double       element;
   Stack_Node*  next;
};

struct Stack_Implementation {
   Stack_Node*  top;
};

Stack create() {
   Stack  s = new Stack_Implementation;
   s->top = EMPTY_STACK;
   return s;
}

// Övriga stackoperationer definieras här...
```

I ett program som ska använder en eller flera stackar inkluderar man filen `Stack.h` och kan sedan deklarera och initiera en stack enligt följande:

```
{
   Stack  s = create();
   int    x;
   …
   push(x, s);
   …
   if (! empty(s))
      x = top(s);
   …
   destroy(s);
}
```

Om `Stack` är implementerad med ett fält eller med en länkad lista påverkar i princip ej användningen av stackvariabeln `s`. Fullständigt oberoende av implementering är dock svårt att uppnå, t.ex. har ju fältet en begränsad storlek (100 element i detta fall) medan den länkade stacken kan växa så länge mer dynamiskt minne kan erhållas. För fältimplementeringen skulle man därför kunna tänka sig att även en operation `full` kunde finnas. En enhetlig hantering är dock möjlig om man använder sig av undantag (exception) för att hantera fallet ”stacken är full”.

Om vi inte använde tekniken att låta datatypen Stack vara en pekare till den post som är själva stacken så skulle vi behöva att definiera den posttypen i inkluderingsfilen i stället, vilket skulle innebära sämre abstraktion.

11 Optimering

En del programmerare lägger ner stor tid på att skriva ”effektiva” program, dvs (oftast) program som går snabbt att köra. Vi brukar kalla detta att (källkods)optimera sitt program. Om man anstränger sig för att skriva snabba program har man infört en extra svårighet, förutom att få programmet att fungera. Programmet blir lätt krångligt och svårläst, eftersom man ofta ägnar sig åt olika ”tricks och fix” för att öka snabbheten.

Det är i de flesta fall onödigt att optimera och även i de fall där det är motiverat är det lätt att man optimerar på fel sätt. Det kan till och med bli så att programmet går långsammare! De ”tricks och fix” man använder för att optimera sina program kanske inte alls fungerar på andra datorer och med andra kompilatorer än just den dator och den aktuella kompilatorn. Om programmet senare ska köras på en annan dator, så kanske det måste skrivas om.

Det finns två grundläggande regler för optimering:

1. Gör det inte!
2. Gör det inte *än*!

Detta betyder att oftast behöver man inte alls optimera. Om man trots allt måste göra det (när programmet faktiskt går för långsamt) bör man vänta i det längsta med optimeringen, så att man verkligen vet *att*, *vad* och *hur* man ska optimera!

Se först till att programmet är korrekt (att det fungerar) och robust (att det klarar även felaktiga och konstiga inmatningar utan att bete sig felaktigt eller krascha), samt att koden är lättbegriplig. *Optimera inte!*

Om du fortfarande tycker att programmet går för långsamt, låt kompilatorn försöka generera effektivare kod. Kompilera programmet med flaggan -O eller motsvarande. *Optimera inte själv, låt kompilatorn göra det!*

Om det i alla fall går för långsamt, kan du börja tänka på att optimera för hand. Gör följande:

1. Ta reda på *var* du ska optimera, dvs var i programmet går det åt mest tid? Det finns analysverktyg som visar hur ofta och hur långvarigt olika delar av ett program används.
2. Optimera på *hög nivå* (algoritmer och datastrukturer). Om du byter en enkel sorteringsalgoritmmot en mer avancerad kanske körtiden minskar avsevärt. Om du byter en enkel sökstruktur mot en hashtabell kan kanske tiden som går åt för sökningar minskas påtagligt.
3. Först i sista hand ska du ägna dig åt lågnivåoptimering, dvs ”tricks och fix”, t.ex. med pekare och bitoperatorer. Sådant kan du göra när du redan har effektiva datastrukturer och algoritmer, programmet ändå är för långsamt och du tagit reda på var det behöver optimeras.

Många ”källkodsoptimerare” brukar skriva om fältkopieringar, som vanligtvis ser ut som

```
int  original[MAXSIZE];
int  copy[MAXSIZE];
...
for (int i = 0; i < n_elements; ++i)
   copy[i] = original[i];
```

till något i stil med

```
int  original[MAXSIZE];
int  copy[MAXSIZE];
...
register int* original_ptr = original;
register int* original_end = original + n_elements;
register int* copy_ptr     = original;

while (copy_ptr < original_end)
   *copy_ptr++ = *original_ptr++;
```

Den senare versionen antar de ska gå snabbare. I realiteten kommer en del kompilatorer att generera exakt samma kod. En del kompilatorer ger faktiskt *långsammare* kod för den senare varianten. Om man betänker att programmet komplicerats i onödan, att det tar längre tid att skriva, att det är större risk för fel, att det är svårare för andra (och dem själva) att förstå, och att tiden för att köra slingan ändå kanske bara utgör en försumbar del av körtiden för hela programmet, kan man undra vad vitsen egentligen är.

Det här betyder inte att man ska strunta i effektivitet. Även ett enkelt program kan göras så långsamt och minneskrävande att det blir helt oanvändbart, om man väljer olämpliga datastrukturer och långsamma algoritmer. Du bör redan från början välja datastrukturer och algoritmer som passar för problemet. Lågnivåeffektiviteten är dock mindre viktig, och hanteras oftast bäst av kompilatorn.

12 Flyttbarhet

Ett flyttbart (portabelt) program är ett som du kan flytta till en annan dator än det skrevs på, med ingen eller liten arbetsinsats. Att skriva program som följer standarderna för C eller C++ är ingen som helst garanti för att programmen är flyttbara. Vissa saker i språken anges uttrycklingen vara inplementationsberoende, andra är inte helt specificerade och kan då variera för olika implementationer. Exempel på sådant som är implementationsberoende:

- namn i inkluderingsdirektiv. Ska du vara riktigt säker: max sex bokstäver (enbart små eller enbart stora), följt av en punkt, följt av *en* bokstav.
- storlek och komplexitet hos program kan begränsas av kompilatorn, t.ex. antal satser som kan nästlas, antal parentesnivåer i uttryck, antal namn som kan deklareras i ett block, antal tecken i namn, antal funktionsparametrar, variablers minnesstorlek, antal värden i en uppräkningstyp, antal tecken i en sträng, radlängd i källkoden, etc. Vanligtvis är dock begränsningarna inte speciellt besvärande, t.ex. kan man ha upp till 32 nästlade parentespar i ett uttryck och skriva upp till 509 tecken på en källkodsrad.
- teckenmängder.
- intern representation för vissa slags värden, t.ex. teckenvärden.
- beräkningsordning i sammansatta uttryck.

Det finns standardinkluderingsfiler där systemberoende värden definieras. Genom att utnyttja sådana i stället för att t.ex. skriva motsvarande literala värden i din kod kan du öka flyttbarheten. I `limits.h` hittar du hur många bitar som ingår i en char, min- och maxvärden för olika tecken- och heltalstyper, hur många tecken som kan hanteras i några olika sammanhang (radlängd, buffertstorlekar, etc.). I `float.h` hittar du min- och maxvärden för olika flyttalstypers, antal siffror i mantissa och exponent, etc.

Kod som du själv skriver och som du vet är implementationsberoende ska du om möjligt hålla isär från annan kod. Det gör det enklare att anpassa koden om programmet ska flyttas till en annan dator. Något som brukar kunna ge kod som ej är flyttbar är sådant som man gör då man lågnivåoptimerar källkod, t.ex. ”tricks och fix” med pekare, bit- och skiftoperationer.

13 Sammanfattning

- Använd meningsfulla och rättvisande namn – undvik förkortningar!
- Använd inte globala variabler i onödan – återanvänd inte variabler!
- Använd symboliska konstanter och namngivna datatyper!
- De viktigaste kommentarerna är de översiktliga och förklarande!
- Kommentera lagom mycket – kommentera rätt saker!
- Använd strukturerad programmering!
- Slingor, **if**-satser och funktioner ska vara korta!
- Tänk på hur du skriver slingor!
- Modularisera och abstrahera – dela upp program i funktioner och på filer!
- Formatera program så att de blir lättlästa!
- Optimera inte förrän det är nödvändigt!
- Tänk på flyttbarhet!
- Följ konventioner!