C – en översikt

Inledning

Programspråket C är ett litet, uttrycksorienterat systemprogrammeringsspråk, som såg dagens ljus 1972. C togs ursprungligen fram för att skriva systemprogram på datorn PDP11. Dess rötter finns i Algol 60 och utvecklingen går att följa via en kedja av systemprogrammeringsspråk: CPL (Combined Programming Language), BCPL (Basic CPL) och B (endast en del av BCPL). B är liksom sina två föregångare ett otypat språk. Steget till C togs genom att lägga till en typstruktur, som hämtades från Algol 68.

C innehåller i stort sett vad som kan förväntas av ett Algolspråk från 70-talets början, vad gäller styrstrukturer, datatyper etc. Däremot är C mer maskinnära än många andra högnivåspråk. Den underliggande maskinen avspeglar sig på olika sätt i språket. En framträdande maskinnära egenskap är möjligheterna att komma åt och manipulera adresser. En annan är vissa operatorers maskinorienterade egenskaper.

Före kompileringen av ett C-program förarbetas källkoden alltid av en *preprocessor*. Preprocessordirektiv kan införas i programmet och dessa behandlas innan programmet kompileras. Enkelt uttryckt rör det sig om textsubstitution av olika slag: inkludering av andra källkodsfiler, substitution av textmakron, villkorlig kompilering i form av exkludering av kod. Anrop av preprocessorn sker automatiskt när program kompileras.

Av historiska skäl är C nära förknippat med operativsystemet UNIX, som idag till allra största delen är skrivet i C. Språket är dock större utanför UNIX-världen. Att C blivit så populärt beror bland annat på att det är en fullgod ersättare för assemblérspråksprogrammering, och att språket finns på en stor mängd datorer. C är också ett språk på modet; ”riktiga” programmerare skriver i C.

C har utformats utan tanke på programmering av stora system. Språket saknar också en del viktiga skyddsnät som moderna högnivåspråk brukar ha. De flesta seriösa användare av C, t ex större företag, brukar ha speciella programmerarhandböcker för sina C-programmerare. Vanligt är att en hel del C-specifika konstruktioner förbjuds helt eller i vissa fall tillåts under vissa premisser, vilket indikerar att det finns en hel del saker i C som innebär risker, eller som man ej bör använda av andra orsaker.

C är numera ANSI-standard, men det är alltid ett problem att i efterhand försöka standardisera ett språk. Det finns tecken som tyder på att C i framtiden kommer att ersättas av språket C++, som är C med en del förbättringar och utvidgningar, främst med klassbegreppet från Simula 67. En annan känd utvidgning av C är Objective C.

Övergripande programstruktur

Det klassiska, första C-programmet är enligt tradition:

```
#include <stdio.h>
main ()
{
   printf("hello world!");
}
```

Programmet, som skriver ut texten `hello world!`, består av ett direktiv till preprocessorn, följt av funktionen `main`.

Direktivet `#include <stdio.h>` innebär att preprocessorn, på den plats där direktivet står, ersätter detta med innehållet i filen `stdio.h`. Filen `stdio.h` är en inkluderingsfil (*header file*), en textfil innehållande definitioner och deklarationer som behövs för att använda fördefinierad standard in- och utmatning. Funktionen `printf` är en sådan fördefinierad funktion.

Funktionen `main` innehåller en sats, funktionsanropet `printf("hello world!")`. Namnet `main` är speciellt. Varje C-program ska innehålla en funktion `main`, i vilken exekveringen startar; ”huvudprogrammet”. Parenteserna `()` efter `main` deklarerar dels att `main` är en funktion, och dels att `main` ej har några parametrar. Parenteserna `{…}` som avgränsar funktionskroppen motsvarar andra språks **`begin`**…**`end`**. Sådana `{…}`-block kan nästas och innehålla både deklarationer och satser, en konstruktion som brukar kallas Algol-block.

Ett C-program består i allmänhet av ett antal funktioner. Funktionen är den enda programenheten i C, men den kan dock användas även som en procedur. Till skillnad från många andra Algolspråk kan ej funktioner nästas. Ett C-program har ur den synvinkeln en flat struktur, som påminner om program skrivna i assemblérspråk eller Fortran. Det finns dock vissa möjligheter att strukturera C-program genom uppdelning av variabel- och funktionsdeklarationer på flera källkodsfiler. Det är också möjlighet att för en global deklaration i en viss källkodsfil bestämma över vilka andra källkodsfiler dess räckvidd ska sträcka sig, genom att i dessa filer antingen direkt skriva eller inkludera en motsvarande externdeklaration. Funktioner på en källkodsfil kan antingen inkluderas i andra källkodsfiler, eller kompileras, kanske hellre, för sig och sedan länkas med övriga programkomponenter.

Ett sätt att strukturera ett C-program är att placera huvudfunktionerna i programmet på separata källkodsfiler, tillsammans med sina logiskt underordnade funktioner. På en gemensam inkluderingsfil, samlas alla globala deklarationer. Om funktioner från fördeklarerade standardbibliotek används, inkluderas även dessa biblioteks inkluderingsfiler i programmets inkluderingsfil. Idén är alltså att ha en gemensam inkluderingsfil för programmets samtliga källkodsfiler. Detta har nackdelen att allt som inkluderas till en viss källkodsfil ej är relevant – “alla inkluderar allt”. Detta är dock det traditionella sättet att strukturera C-program.

Ett annat sätt att dela upp ett program, är att försöka efterlikna moduler, typ paket i Ada. Uppdelningen av funktioner på källkodsfiler kan vara densamma som i förra fallet. Skillnaden ligger i att varje källkodsfil här har sin egen inkluderingsfil, och att varje källkodsfil endast inkluderar inkluderingsfiler för de “moduler” som den verkligen använder. Detta sätt att dela upp ett program ger ofta en bättre och mer logisk programstruktur.

Kompilatorns kontroll av gränsytor mellan funktioner är i många fall obefintlig eller bristfällig. Det brukar dock i många C-system finnas ett program som heter Lint, som gör mer omfattande kontroller. Den nya standarden innebär också vissa förbättringar på den här punkten, exempelvis genom den från C++ anammade prototypformen för att skriva funktionshuvuden.

Lexikala aspekter

C är ett språk som anses kunna ge ”kryptiska” program. Det beror till viss del på språkets lexikala utformning. Många specialtecken används i C där andra språk använder ”klartext”, vilket onekligen leder till en viss ”kryptiskhet”. Språkets många operatorer och den uttrycksorienterade syntaxen förstärker det intrycket. Naturligtvis beror också resultatet på programmerarens användning av språket.

Identifierare får bestå av bokstäver, siffror och understrykningstecknet. Det första tecknet får ej vara en siffra, men väl ett understrykningstecken. Notera att stora och små bokstäver är distinkta, vilket ej är så vanligt i programspråk. Det är vanligt att det förekommer både inledande och avslutande understrykningstecken i en del fördefinierade namn, exempelvis följande makron i ANSI-standarden:

`__DATE__`	aktuellt datum
`__TIME__`	aktuell tid
`__STDC__`	talet 1, visar att realiseringen följer ANSI-standarden.

Detta är gjort för att undvika namnkollisioner. I egna namn bör man alltså av den anledningen, liksom av läsbarhetsskäl undvika inledande och avslutande understrykningstecken.

Följande reserverade ord definieras i ANSI-C:

```
auto      break    case      char     const     continue  default
do        double   else      enum     extern    float     for
goto      if       int       long     register  return    short
signed    sizeof   static    struct   switch    typedef   union
unsigned  void     volatile  while
```

Kommentarer inleds med `/*` och avslutas med `*/` och kan i stort sett placeras var som helst.

Teckenlitteraler omges av apostrofer:

```
'A'    'p'    '+'    '9'
```

Till teckenlitteraler räknas också de s k ”escapesekvenserna”, av vilka det finns fyra skrivbara:

```
'\''    '\"'    '\?'    '\\'
```

Tecknet \ anger att det efterföljande tecknet ska skrivas ut och ej ges den speciella tolkning det normalt har i en teckenliteral och även i en strängliteral. Det finns också ”escapesekvenser” för icke skrivbara tecken:

`\0`	Tomtecknet	`\n`	Nyradstecknet
`\a`	Ljud- eller ljussignal	`\r`	Vagnreturtecknet
`\b`	Backstegstecknet	`\t`	Horizontaltabuleringstecknet
`\f`	Sidframmatningstecknet	`\v`	Vertikaltabuleringsteckne

Dessutom finns ”escapesekvenser” för att ange teckenkoder, som exempelvis `'\027'`, samt den märkliga möjligheten att i en teckenliteral skriva flera tecken, exempelvis `'ABC'`, vilket dock är ett värde av typen **`int`**. Stränglitteraler avgränsas med citationstecken:

```
"Detta är en strängliteral"
```

Notera att en sådan literal representeras av en pekare i det sammanhang den förekommer, en pekare som refererar det minnesutrymme där teckensekvensen lagras.

Numeriska litteraler i C är icke-negativa. Heltalslitteraler kan vara decimala, oktala (`0`...) eller hexadecimala (`0x`... eller `0X`...), och är normalt av typ **`int`**. Flyttalslitteraler är alltid decimala, och normalt av typ **`double`**, men kan anges som **`float`** (`f` eller `F`) eller long **`double`** (`l` eller `L`). Några exempel:

```
1          45          4711          Decimala
0          07          0494          Oktala
0x1        0XFF        0x1A          Hexadecimala
0.1       .005         2.e-5        double
0.707f     6.023e23F                 float
64.5l      1.E99L                    long double
```

Datatyper, deklarationer och lagringsklasser

I C finns heltalstyper, flyttalstyper, uppräkningstyper, pekartyper, fälttyper, posttyper, bitfält, och variabla typer. Språket är svagt typat. En hel del automatiska typomvandlingar gäller och uttrycklig typomvandling kan ske fritt. Någon egentlig möjlighet att definiera egna, distinkta datatyper finns ej. Lagringsklasser anger hur variabler ska hanteras minnesmässigt.

Variabeldeklarationer

Variabler kan införas antingen utanför funktioner, ”globalt”, eller i block:

```
int  top = 0;  /* "Global" variabel */

void fun (int i)
{
  float  x;  /* Lokal variabel i funktionen */
  …
}
```

Variabeln `top` är en globalt deklarerad heltalsvariabel. Den kan refereras inifrån funktionen `fun`, liksom eventuella andra funktioner som finns. Parametern `i`, och variabeln `x` lokala variabler i funktionen `fun`.

Variabler som deklareras i block ska deklareras före satserna i varje block och kan med vissa begränsningar ges initialvärden i samband med deklarationen. Några exempel på variabeldeklarationer:

```
char   ch = ' ';
int    i, j = 0;
float   x = 1.95e3;
```

Datatypen anges först, sedan ett eller flera variabelnamn, och om så önskas initialvärden. Notera att initieringen på **`int`**-raden endast gäller variabeln `j`.

Konstanter

Konstanter kan i C deklareras enligt:

```
const pi = 3.1415;
```

Denna möjlighet har införts i ANSI-standarden. Det går att göra liknande genom preprocessormakron.

Numeriska typer

Numeriska typer är heltalstyper och flyttalstyper. Till heltalstyperna räknas typerna **`char`**, **`short int`**, **`int`** och **`long int`**. För dessa är det dessutom möjligt att ange **`unsigned`** eller **`signed`**, dvs om den binära koden ska tolkas som innehållande en teckenbit eller ej. Typen **`char`** är naturligtvis avsedd för att hantera tecken, men notera att ibland måste **`int`** används i stället vid teckenhantering, och för övrigt gäller att värden av typ **`char`** alltid omvandlas till **`int`** i uttryck. Att skriva enbart **`short`** är detsamma som att skriva **`short int`**. Denna princip gäller även för **`long`** och **`unsigned`**. Typen **`int`** är ”normaltypen” bland heltalstyperna, och även i andra sammanhang förutsätts **`int`** då inget annat anges.

Flyttalstyperna är **`float`**, **`double`** och **`long double`**. I äldre C-system förekommer **`long float`** i stället för **`double`**. På samma sätt som **`int`** är normaltypen för heltal, så är **`double`** normaltypen för flyttal.

Fält

Fält deklareras genom att ange elementtyp och antalet element i varje dimension. Deklarationen:

```
int  iarr[20];
```

inför en fältvariabel med 20 element av typ **`int`**. Förstaindex är alltid 0, så indexintervallet för de 20 elementen i `iarr` är `0..19`, vilket kan störa tankebanorna litet; indexvärden och ordningstalen för elementen skiljer med 1. Ett indexuttryck, som exempelvis `iarr[i]` i tilldelningssatsen:

```
j = iarr[i];
```

innebär att indexoperatorn `[]` appliceras på minnesadressen (pekaren) `iarr` och indexvärdet `i`.

Pekare

Pekare i C är typade, dvs en viss pekartyp bör endast användas för att peka på objekt av en viss typ. Det finns dock ingen kontroll av att så verkligen är fallet, och detta är speciellt viktigt att beakta i samband med adressaritmetik. Deklarationen:

```
int  *ip;
```

betyder att variabeln `ip` är en pekare till **`int`**. Genom en tilldelning som:

```
ip = iarr;
```

kommer heltalspekaren `ip` att tilldelas adressen till första elementet i `iarr.` Fältnamn som `iarr` representerar en pekare till det första elementet i fältet.

Adresser, pekare och fält

Det finns två operatorer för att hantera minnesadresser, eller pekare: & och *. Med adressoperatorn &, kan adressen till en variabel eller till en funktion erhållas.

```
int  i, j, *ip;

ip = &i;
j = *ip;
```

Variablerna `i` och `j` är heltalsvariabler, `ip` är en pekarvariabel som kan peka på objekt av typ **`int`**. Operatorn & medför att den ”dereferencing” som i de flesta fall görs av ett variabelnamn som står i högerledet av ett uttryck, *ej* görs. Variabeln `ip` tilldelas i stället den minnesadress som variabelnamnet `i` representerar. Notera att det är fullt tillåtet att skriva:

```
ip = i;
```

men det är ej adressen till variabeln `i` som erhålls, utan heltalsvärdet i variabeln. Det värdet kan sedan komma att tolkas som en adress när `ip` används. Detta är bara ett exempel på faror som lurar i C. Språket har definitivt sina risker till följd av den ”fria” adresshanteringen, svag typning, automatiska typomvandlingar och kryptisk syntax, i kombination med språkets uttrycksorienterade karaktär.

Det har naturligtvis ingen direkt betydelse för själva pekarvariabeln, om den ska peka på ett heltal eller på något annat, men när en pekarvariabel används i uttryck, är det i många fall mycket väsentligt för C-kompilatorn att känna till vad för slags dataobjekt pekaren är tänkt att referera, för att korrekt kod ska genereras.

Operatorn * är ”dereferencing”-operatorn, dvs motsatsen till adressoperatorn &. När *-operatorn appliceras på en pekare (en minnesadress):

```
j = *ip;
```

ger den som resultat det värde som finns lagrat i minnesadressen. Satserna:

```
ip = &i;   /* ip <– adressen till i   */

j = *ip;   /* j <– värdet i adress ip */
```

innebär alltså detsamma som att skriva `j = i`. Pekare och fält är mycket nära relaterade i C. Alla operationer som kan göras genom indexering kan nämligen också göras med adressaritmetik:

```
j = iarr[i];
```

kan uttryckligen skrivas enligt adresseringsmekanismen på maskinnivån, på formen *basadress+offset*:

```
j = *(iarr + i);
```

Detta tolkas enligt följande. Värdet av `i`, underförstått multiplicerat med elementstorleken för `iarr`, adderas till adressen `iarr` (eftersom `iarr` är ett fält). Därefter appliceras *-operatorn på det erhållna adressvärdet, och som resultat erhålls en heltalstolkning av värdet i adressen `iarr+i`, vilket slutligen tilldelas variabeln `j`.

Adressen till godtyckliga variabler kan erhållas med adressoperatorn &, liksom även komponenter i strukturerade typer. Ett fält är ju inget annat än en samling variabler under gemensamt namn. Adresser till element i `iarr` kan erhållas enligt:

```
ip = iarr;      /* Första elementet */

ip = &iarr[0];  /* Första elementet */

ip = &iarr[5];  /* Sjätte elementet */

ip = &iarr[20]; /* Går, men utanför iarr */
```


Det 6:e elementet i `iarr` kan kommas åt på flera sätt. Några exempel:

```
i = iarr[5];     /* ”Normal” indexering */

i = *(iarr + 5); /* Adressaritmetik och ”dereferencing” */

ip = iarr;       /* Sätt pekare på första elementet i iarr */

i = *(ip + 5);   /* Pekararitmetik och ”dereferencing” */

ip = ip + 5;     /* Räkna upp pekaren till element 6 i iarr */

i = *ip;         /* 4 */
```

Det är i samtliga dessa fyra fall viktigt att `ip` är deklarerad som pekare till **`int`**, för att 5:an i de olika uttrycken ska skalas korrekt. Om **`int`**-värden lagras i 4 bytes, så ska 5:an multipliceras med 4.

Av effektivitetsskäl är det vanlig att använda pekararitmetik i stället för indexering, för att stega igenom fältstrukturer. Följande två exempel visar en ”traditionell” lösning och en typiskt C-lösning för att stega igenom `iarr`:

```
for (i = 0; i < 20; i++)
  printf("%d", iarr[i]);

for (ip = iarr; ip < &iarr[20]; ip++)
  printf("%d", *ip);
```

`for`-slingorna är likvärdiga beräkningsmässigt: en enkel styrvariabel tilldelas ett startvärde, och för varje varv görs sedan ett test mot ett konstant slutvärde och en uppstegning av variabeln. `i++` motsvarar stegning `i=i+1`, `ip++` motsvarar `ip=ip+ sizeof(int)`. Operatorn **`sizeof`** ger oss minnesbehovet för angiven datatyp, variabel eller uttryck. **`sizeof`** är alltid ett konstant värde, som fixeras vid kompileringen.

Skillnaden ligger i uttrycken `iarr[i]` och `*ip`. Det senare uttrycket är enbart en ”dereferencing”, medan indexeringen innebär beräkningsarbetet:

```
*(iarr + i * sizeof(int))
```

En multiplikation (värst), en addition, samt en ”dereferencing”; `iarr` och **`sizeof`**(**`int`**) är konstanter.

Pekare i C har, som framgått ovan, en betydligt bredare användning än pekare i de flesta andra språk, där de endast används för dynamisk minneshantering.

Uppräkningstyper

Uppräkningstyper, **`enum`**, har införts i C efter den ursprungliga definitionen. Deklarationen:

```
enum Color {green, red, blue, yellow}  shade;
```

inför en variabel `shade` av typen **`enum`** `Colour`. Åtminstone i ANSI-standarden kommer uppräkningstyper att vara att betrakta som ytterligare en heltalstyp. Uppräkningslitteraler representeras ju med heltal på maskinnivå.

Posttyper

Poster kallas i C för ”structures”, **`struct`**. Deklarationen:

```
struct Complex {
  double   real, imag;
} c1, c2;
```

inför en poststruktur med två fält, `real` och `imag`, av typen **`double`**. Samtidigt deklareras två postvariabler, `c1` och `c2`, av typen **`struct`** `Complex`. Deklarationen:

```
struct Tree {
  int            key;
  struct Tree  *left;
  struct Tree  *right;
} *root;
```

inför en poststruktur med tre fält, `key` av typen **`int`**, samt `left` och `right` som är pekare till **`struct`** `Tree`. Samtidigt deklareras en variabel `root` som är en pekare till **`struct`** `Tree`.

Bitfält är poststrukturer där utläggningen av fälten specificeras på bitnivå. Deklarationen:

```
struct Op_Code {
  unsigned   instr   : 3;
  unsigned   address : 5;
};
```

anger att för variabler av typ **`struct`** `Op_Code` upptar fältet `instr` 3 bitar, och `address` 5 bitar.

Variabla typer

Variabla typer, **`union`**, inför objekt som kan anta värden av olika typ. Deklarationen:

```
union Data {
  char     c;
  int      i;
  double   d;
} x;
```

inför en variabel `x` av typen **`union`** `Data`, som kan anta ett värde som antingen är av antingen typ **`char`**, eller av typ **`int`**, eller av typ **`double`**. Vilken typ det aktuella värdet för `x` har ”håller `x` själv reda på”. Detta kan dock ej kontrolleras. För att hålla reda på sådan information måste hjälpvariabler införas. Genom att införa komponenter av typen **`union`** i posttyper, kan motsvarigheten till andra språks variantposter erhållas:

```
enum node_type  (internal, terminal);

struct Tree {
  enum node_type  tag;
  union {
    char    operator;  /* internal */
    double  operand;   /* terminal */
  }
  struct Tree  *left, *right;
} *root;
```

Void

Det finns en speciell ”typ” **`void`**, som används för att antingen markera att en funktion ej returnerar något värde:

```
void push (item);
```

eller att en funktion ej har några parametrar:

```
int rand (void);
```

eller för att deklarera pekarvariabler som får peka på godtyckliga dataobjekt:

```
void  *upa;
```

Däremot kan ej variabler av typen **`void`** deklarera.

Typdeklarationer

Observera att deklarationer som visats ovan, t.ex.

```
enum   Colour {…}

struct Complex {…}

struct Tree {…}
```

ej innebär att namnen `Colour`, `Complex` eller `Tree` införs som datatyper, utan exempelvis `Tree` är så att säga enbart ett namn på den efterföljande postbeskrivningen. Vill man senare införa ytterligare postpekare skriver man:

```
struct Tree  *t1, *t2;
```

Egna ”typnamn” i C kan dock deklareras genom **`typedef`**. Det är dock ej lika användbart som i vissa andra språk, eftersom **`typedef`** ej inför distinkta typer, t.ex. innebär deklarationen

```
typedef int  Boolean;
```

i och för sig att `Boolean` kan användas som ett datatypnamn för att deklarera variabler

```
Boolean  error_flag;
```

men variabeln `error_flag` kommer att vara en helt vanlig **`int`**. Detta är vad standarden föreskriver, men kompilatorer kan naturligtvis ändå varna för blandade uttryck. Skulle man vilja införa en ”datatyp” för **`struct`** `Tree` kan det se ut som:

```
typedef struct Tree {
  int           key;
  struct Tree  *left, *right;
} TREE;

TREE  *root;
```

Skälen till att använda **`typedef`** är dels ökad läsbarhet som ovanstående exempel åskådliggör, dels kan det förbättra flyttbarheten för program, ofta i kombination med preprocessordirektiv som:

```
#ifdef VAX
  typedef unsigned char   SMALL_COUNTER;  /* 0 .. 255 */
#endif
#ifdef TOPS20
  typedef unsigned short  SMALL_COUNTER;
       /* 7-bit chars, must be short */
#endif
```

Definitioner och deklarationer

Variabler kan deklareras antingen utanför funktioner, på ”filnivå”, eller inuti funktioner, lokala variabler. Funktioner kan ej nästlas i varandra, så det finns bara en funktionsnivå i C-program.

I C används begreppen *definition* och *deklaration* i samband med införandet av variabler och funktioner. En definition är alltid också en deklaration, medan det omvända ej gäller. En variabeldefinition är den deklaration som ursprungligen inför variabeln i ett program och som är orsaken till minnestilldelningen. Andra deklarationer av variabeln anger bara dess existens. När man i en källkodsfil vill använda en variabel som är definierad i en annan källkodsfil ska variabeln externdeklareras.

Deklarationer på funktionsnivå

Variabeldefinitioner eller andra definitioner som ska vara synliga för flera funktioner i ett C-program införs på samma nivå som funktionerna, på filnivå. Det finns två typer av sådana variabler, dels de som

kallas *global* och dels de som kallas **static**. Gemensamt har de egenskapen att de är tilldelade minne under hela programexekveringen. Det som skiljer är räckvidden i programmet. Ett exempel på en definition av en global heltalsvariabel `Global_Flag`:

```
int  Global_Flag;
```

Variabeln `Global_Flag`:s räckvidd är hela programmet, men synligheten är begränsad till definitionsfilen om ej **extern**-deklarationer gör den synlig i andra filer. En externdeklaration ser ut som:

```
extern int  Global_Flag;
```

Om räckvidden för en variabel som definieras på filnivå ska begränsas, kan detta göras genom att deklarera variabeln som **static**:

```
static int  File_Flag;
```

sådan variabels livstid är också densamma som hela programmets livstid, men räckvidden är garanterat begränsad till definitionsfilen.

Deklarationer i block

Variabler deklarerade i block lagras antingen på exekveringsstacken, i **register**, eller i någon permanent dataarea, beroende på hur de deklareras. Det finns fyra olika typer av deklarationer i block: **auto**, **register**, **static** och formella parametrar. Alla deklarationer i ett block måste placeras före den första exekverbara satsen. Normala synlighetsregler gäller.

Auto är normalmoden för variabler deklarerade i block. De existerar endast under exekveringen av blocket:

```
{
  auto int x;
  …
}
```

Nyckelordet **auto** är valfritt och används sällan i praktiken.

Registervariabler

Registervariabler lagras om möjligt i ett **register** för att snabb kunna läsas och ändras. Om det ej finns något **register** tillgängligt placeras de i stället på stacken, precis som **auto**-variabler:

```
{
  register int  y;
  …
}
```

Naturligtvis måste **register**-deklarerade variabler vara av sådan typ att de kan lagras i ett maskinregister. Adressoperatorn & kan av naturliga skäl ej användas på registervariabler.

Funktionsparametrar

Funktionsparameternamn införs i funktionshuvudet inom parenteser efter funktionsnamnet. Deras typ kan deklareras på två sätt i den nya standarden. Det traditionella sättet är att skriva en parameternamnslista i funktionshuvudet, och ha separata typdeklarationer innan själva funktionskroppen:

```
int func (x, y)
register int  x;
double  y;
{
  …
}
```


Även funktionsparametrar kan alltså deklareras som **register**. I ANSI- standarden finns också ett s k prototypformat för deklaration av funktionsparametrar, direkt inspirerad av C++ men vanlig i många andra språk:

```
int func (register int x, double y)
{
  …
}
```

I C finns bara en parametermod, värdeanrop (”call by value”). Mer om detta nedan.

Static-variabler

static-deklarerade variabler i block avviker från övriga blockvariabler genom att deras minnesutrymme existerar under hela programexekveringen (”retained variables”), och därmed bibehålls också deras värde mellan olika aktiveringar av blocket:

```
{
  static  int  flag;
  …
}
```

Block med denna typ av variabler blir *historikkänsliga*. Vilken som helst blockvariabel utom formell funktionsparameter kan deklareras som **static**. Motsvarar **own**-deklarerade variabler i Algol60.

Volatile-variabler

Dataobjekt som lagras i speciella minnesceller, vilkas innehåll kan ändras utanför programmets kontroll, kan i ANSI-standarden deklareras som **volatile**. Detta innebär att ingen optimering får göras vid kompilering, utan objektprogrammet ska följa källprogrammet till punkt och pricka när sådana objekt bearbetas. Detta är tex viktigt att kunna styra då data kan skrivas i register eller andra minnesadresser av processer utanför programmets kontroll.

Initiering av variabler

Variabler kan, med vissa undantag, initieras i samband med deklarationen. Permanenta variabler initieras automatiskt till tomtecken `(0, NULL, '\0')`, om inget annat anges. Varje konstantuttryck kan användas som initieringsvärde. Enkla typer:

```
int   i = 1;

float x = 3.1415e-2;
```

Fält initieras med fältkonstanter (”*aggregate*”):

```
char s[10] = { 'A','B','C' };

int  a[]   = {1,2,4,8,16,32};
```

Om dimension anges sätts ej listade element till 0. Om ingen dimension anges bestäms storleken av antalet värden i fältkonstanten. Strängar kan initieras på två sätt, antingen med en strängliteral eller med en fältkonstant:

```
char s[] ="Hello";

char s[] = {'H','e','l','l','o','\0' };
```

I det senare fallet måste det avslutande tomtecknet, `'\0'`, anges. I det förra fallet sätter kompilatorn in det automatiskt. En sträng ska alltid avslutas med ett tomtecken. Många fördefinierade funktioner använder detta. När ej antalet element för ett fält anges i en deklaration måste ett initialvärde anges. Då bestämmer initialvärdet antalet element i fältobjektet (6 i exemplen närmast ovan).

Poster kan initieras med postkonstanter (”*aggregate*”):

```
struct person {
  int   height;
  char  gender;
}

struct person  x = { 70, 'Y' };

struct person  family[] = {
  { 73, 'X' }
  { 68, 'Y' }
  { 50, 'X' }
}
```

I det senare exemplet avgör alltså postkonstanten att fältet `family` kommer att innehålla tre poster.

Fältkonstanter behöver ej vara uttömmande. I vissa fall behöver inte ens parenteser för delfält sätta ut då fältkonstanter nästas i fältkonstanter.

Operatorer och uttryck

C är påfallande rikt på operatorer, och en del språkkonstruktioner som i många andra språk ej brukar ses som operatorer, ses uttryckligen som sådana i C. Exempel på sådana är:

- postfixoperatorn `()` som innebär ett funktionsanrop:

  ```
  x = sin(x)
  ```

- postfixoperatorn `[]` som innebär indexering i en fältstruktur:

  ```
  x = a[i]
  ```

- infixoperatorn `.` (punkt) för att välja en komponent i en poststruktur:

  ```
  x = p.name
  ```

- infixoperatorn `->` för att välja en komponent i en poststruktur som refereras av en pekare:

  ```
  x = pp->name
  ```

Aritmetiska operatorer

De aritmetiska operatorerna är:

```
+    -    *    /    %    ++    --
```

De fyra första står för addition, subtraktion, multiplikation och division. Observera att om båda operanderna till divisionsoperatorn är heltal, görs en heltalsdivision. Den femte operatorn (`%`) är modulooperatorn som ger resten vid heltalsdivision. De två sista (`++` och `--`) är upp- respektive nedräkningsoperatorerna, som är litet speciella. Motsvaras delvis av ”succ”- och ”pred”-funktioner i andra språk.

Upp- och nedräkningsoperatorerna är enställiga operatorer med bieffekt, dvs de ändrar värdet på sin operand. Dessutom kan de sättas antingen före eller efter sin operand, vilket har betydelse då dessa ingår i sammansatta uttryck. Sätts operatorn före sin operand, görs upp- eller nedräkningen först och därefter används operandens (nya) värde i den fortsatta beräkningen. Om `i` har värdet 1 innebär satsen:

```
j = ++i;
```


att i först räknas upp till 2, sedan tilldelas y detta värde. Observera att i har ändrat värde till 2. Sätts operatorn efter sin operand, använd operandens (gamla) värde först, sedan räknas den upp eller ned. Om `x` har värdet 5 innebär satsen:

```
y = x--;
```

att först tilldelas `y` värdet 5, sedan stegas `x` ned till 4. Om man vill åstadkomma enbart en stegning av en variabel, kan uttrycket skrivas som en uttryckssats:

```
i++;
```

I det här fallet har det ingen betydelse om operatorn sätts före eller efter operanden.

Tilldelningsoperatorer

I de flesta språk förekommer normalt endast en symbol för tilldelning, i C finns 11 stycken tilldelningsoperatorer:

```
=    +=   -=    *=    /=    %=    >>=    <<=    &=    ^=    |=
```

Den första (`=`) är den ”vanliga” tilldelningsoperatorn. De övriga innebär en kombination av en operation och en efterföljande tilldelning. Uttrycket:

```
x += 1
```

innebär detsamma som:

```
x = x + 1
```

Denna typ av operatorer gör det möjlig för kompilatorn att lägga ut effektivare kod, genom att dubbelberäkningar kan undvikas. I följande uttryck beräknas de två identiska indexuttrycken:

```
a[m + i, n + j] = a[m + i, n + j] + i
```

I följande uttryck undviks detta:

```
a[m + i, n + j] += i
```

Det är vanligt att programvaruföretag förbjuder användning av tilldelningsoperatorer utöver `=`.

Relationsoperatorer

Relationsoperatorerna är de sex vanliga för test på likhet, olikhet, mindre-än, mindre-än-eller-lika-med, större-än och större-än-eller-lika-med:

```
==    !=    <    <=    >    >=
```

Booleoperatorer

Booleoperatorerna opererar på logiska värden, *sant* och *falskt*. I C finns ingen specifik booletyp, utan logiska värden representeras som heltal, standardmässigt 0 för falskt och 1 för sant, men alla värden som kan tolkas som ett numeriskt värde skilt från noll tolkas som sant i uttryck. De tre booleoperatorerna för logisk negation, logiskt och, samt logiskt eller är:

```
!    &&    ||
```

Notera att `&&` och `||` egentligen fungerar som ”and then” respektive ”or else”, dvs för `&&` beräknas högeroperanden endast om vänsteroperanden är sann, och för `||` beräknas högeroperanden endast om vänsteroperanden är falsk. I annat fall kan hela uttryckets värde bestämmas ur vänsteroperandens värde.

Bitoperatorer

Bitoperatorer finns för 1-komplement, bitskift höger eller vänster, bitvis och, bitvis eller, samt bitvis exklusivt-eller:

```
~    >>    <<    &    |    ^
```

För skiftoperatorerna gäller att vänsteroperanden skiftas det antal bitar som högeroperanden anger. Uttrycket:

```
i >> 5
```

innebär alltså att bitarna i variabeln `i` skiftas 5 steg åt höger. Ett vänsterskift är alltid logiskt, dvs nollor skiftas in från höger. När det gäller högerskift så gäller att om den variabel som skiftas är **`unsigned`**, så gäller logiskt skift, dvs nollor skiftas in från vänster. Om däremot variabeln är **`signed`**, t ex en vanlig **`int`**, så kan skiftet vara antingen logiskt eller aritmetiskt. Det sistnämnda innebär att teckenbiten avgör vad som skiftas in från vänster. Vilket som gäller beror på den aktuella C-kompilatorn, det är valfritt att realisera högerskift som logiskt eller aritmetiskt.

Adressoperatorn & och ”dereferencingoperatorn” *

En litet ovanlig egenskap hos C, jämfört med många andra högnivåspråk, är möjligheten att hantera adresser, och framför allt möjligheten att begära adressen även för ett icke-dynamiskt dataobjekt. Det finns två operatorer som kan sägas höra ihop i detta avseende, adressoperatorn och dess ”invers”, & och *. Båda är enställiga prefixoperatorer och förekommer i samband med ”vanlig” dynamisk minneshantering, men även i samband med överföring av funktionsparametrar och i samband med hantering av fält. Många exempel gavs redan i avsnittet om datatyper.

Fältoperatorn [ ]

Indexering i C utförs med fältoperatorn `[]`, en enställig postfixoperator som står efter sin operand:

```
p[i]
```

Notera att detta är ekvivalent med att skriva `*(p+i)`. Skalning med hänsyn till elementens storlek sker automatiskt. Exempel gavs redan i avsnittet om datatyper.

Operationer på poster och variabla typer

För att referera ett fält i en poststruktur eller en medlem i en **`union`** finns två operatorer, `.` (punkt) och `->`. Punktvarianten används för ”vanliga” objekt, medan pilvarianten används för objekt refererade av pekare:

```
{
  struct Complex {
    float real, imag;
  } c, *pc;

  c.real = 0.0;
  pc = &c;
  pc->imag = 0.0;
}
```

Villkorsoperatorn ? :

Villkorsoperatorn är också ett litet exotisk inslag i C. Den är treställig, dvs ska ha tre operander, och är en motsvarighet till **`if`**-satsen, men på uttrycksform. Formellt ska ett villkorsuttryck ha formen:

e ? e_T : e_F

Om uttrycket e beräknas till sant, så beräknas uttrycket e_T och dess värde bestämmer värdet för villkorsuttrycket, annars om e beräknas till falskt beräknas uttrycket e_F och dess värde bestämmer värdet för hela villkorsuttrycket. Uttrycket:

```
max = (x > y) ? x : y
```

motsvarar alltså **`if`**-konstruktionen:

```
if (x > y) max = x; else max = y;
```

Villkorsoperatorn hör till det som brukar förbjudas.

Kommaoperatorn ,

Ett kommauttryck har formen:

e_1, e_2

Först beräknas uttrycket $e1$, sedan uttrycket $e2$. Värdet som beräkningen av uttrycket e_1 ger ignoreras, och kommauttryckets värde ges enbart av värdet av uttrycket $e2$. Kommaoperatorn har alltså enbart funktionen av att sekvensera en följd av uttryck, den kombinerar på inget sätt värdena av sina operander. Uttrycket $e2$ kan i sin tur vara ett kommauttryck osv, så godtyckligt många uttryck kan radas upp i ett sammansatt kommauttryck. Kommaoperatorn kommer främst till användning där syntaxen säger att det får stå ett uttryck men där man vill kunna ha flera. Detta gäller t.ex. ofta i **`for`**-satsen:

```
for (i = 0, j = 1; i < max; i++, j++) …;
```

Ett semikolon efter `i = 0` skulle komma i konflikt med **`for`**-satsens syntax. Kommaoperatorn brukar förbjudas i programvaruföretagens programmerarhandböcker.

Operatorn `sizeof`

Operatorn **`sizeof`** är en god hjälp för att skriva flyttbar kod. I stället för att använda maskinspecifika minnesstorlekar för olika datatyper, variabler eller uttryck används **`sizeof`**:

```
sizeof(e)
```

ger det antal bitgrupper (bytes) som behövs för att lagra ett värde av e:s typ. e kan ange ett objekt, ett värde eller en typ.

Notera, att om e är ett uttryck, som exempelvis i:

```
… sizeof(x + i++) …
```

så kommer detta ej att ge upphov till någon kod som exekveras. Kompilatorn kontrollerar endast vilket minnesbehov som resultatvärdet av `x + i++` kräver.

Typomvandling

Uttrycklig typomvandling kallas ”*cast*” och har formen:

```
(typnamn) e
```

detta omvandlar värdet av uttrycket *e* till angiven typ. Omvandling från **`int`** till **`double`**, `i` är en variabel av typ **`int`**, `d` är en variabel av typ **`double`**, ser ut som:

```
d = (double) i
```

Funktionsoperatorn ( )

Anropa funktion *f* med argument *e*1, *e*2, …, *e*n:

```
f(e1, e2, ...,en)
```

Operatorprioritet och associativitet

Det finns 15 prioritetsgrupper för operatorer:

()	[]	.	->	++	--						(H->V)
!	~	-	++	--	&	*	(*typnamn*)	**sizeof**			(H->V)
*	/	%									V->H
+	-										V->H
<<	>>										V->H
<	<=	>	>=								V->H
==	!=										V->H
&											V->H
^											V->H
|											V->H
&&											V->H
||											V->H
?:											H->V
=	*=	/=	%=	+=	-=	<<=	>>=	&=	^=	|=	H->V
,											V->H

I frånvaro av parenteser används associativitetsregler för att gruppera operander då operatorerna är i samma prioritetsgrupp. Operatorerna `&&`, `||`, `?:`, och `,` (komma), garanterar att operanden längst till vänster *alltid* beräknas först. För övriga operatorer varierar beräkningsordningen mellan kompilatorer. Vilken associativitet som gäller, vänster-till-höger (V->H) eller höger-till-vänster (H->V) anges till höger i tabellen på nästa sida. Notera att C- kompilatorn i vissa fall kan sätta sig över även parenteser, exempelvis kan (a + (b + c)) komma att beräknas ((a + b) + c).

Aritmetiska typomvandlingar

För aritmetiska typer finns följande automatiska typomvandlingar, *"coercions"*, definierade för uttryck:

1. **`char`** och **`short int`** omvandlas till **`int`**, **`unsigned char`**, **`unsigned short`** till **`unsigned int`**.
2. Om en av operanderna är **`double`**: andra operanden omvandlas till **`double`**, som även är resultatets typ.
3. I annat fall, om någon operand är **`unsigned long`**: andra operanden omvandlas till **`unsigned long`**, som även är resultatets typ.
4. I annat fall, om någon operand är **`long`**: den andra omvandlas till **`long`**, som även är resultatets typ.
5. I annat fall, om en operand är **`long`** och den andra **`unsigned int:`** båda omvandlas till **`unsigned long`**, som även är resultatets typ.
6. I annat fall, om ena operanden är **`unsigned`**: den andra omvandlas till **`unsigned`**, som även är resultatets typ.
7. I annat fall måste bägge operanderna vara **`int`**, och detta är också resultatets typ.

Satser

Program skrivs på fritt format. Strukturerade satser kan nästlas. En sammansatt sats skrivs på formen:

```
{
  satser
}
```

ett block (”Algolblock”) på formen:

```
{
  deklarationer
  satser
}
```

Uttryckssatser

Uttryck blir i C satser då de följs av ett semikolon. Exempelvis kan uttrycket `i++` skrivas som satsen:

```
i++;
```

vilket är meningsfullt om man vill räkna upp variabeln `i` men ej är intresserade av uttryckets värde.

Villkorssatser

Villkorssatser finns av två slag, **`if`** och **`switch`**. **`if`**-satsen har två former, utan och med **`else`**:

```
if (fp = NULL) exit(1);

if (x > y)
  max = x;
else
  max = y;
```

Om fler än en sats finns i grenarna, så måste sammansatt sats införas.

`switch`-satsen är en primitiv, mer maskinnära form av andra språks case-sats:

```
switch (c) {
  case ' ' :
  case '\t' :
  case '\n' :
    white_space++;
    break;
  default :
    others++;
    break;
}
```

Varje **`case`**-alternativ kan bara innehålla ett konstant värde. Om flera värden ska dela kod, görs detta som i exemplet ovan. Man radar upp flera satslösa **`case`**-alternativ efter varandra; `' '`, `'\t'` och `'\n'` behandlas samtliga i satserna efter **`case`** `'\n'`, till följd av s k ”fall-through” i de två första alternativen. Det krävs en **`break`**-sats för att lämna **`switch`**-satsen innan den är slut. Den sista **`break`**-satsen är ej nödvändig, men brukar ändå införas. **`switch`**-satsen är en potentiell källa till mindre uppenbara fel, till följd av oavsiktliga ”fall through”.

Repetitionssatser

Det finns tre former av repetitionssatser, **while**, **do**-**while** och **for**:

```
while ((c = getchar()) != '\n')
  putchar(c);
putchar(c);

do {
  c = getchar();
  putchar(c);
} while (c != '\n');

for (sum = 0, i = 0; i < n; i++)
  sum += iarr[i];
```

while- och **do**-**while** är villkorsstyrda slingor, förtestad respektive eftertestad. I samtliga fall, även som synes i **do**-**while**, krävs en sammansatt sats om fler än en sats ska repeteras.

I C är **for**-satsen den mest generella repetitionssatsen, betydligt mer generell än vad **for**-satser brukar vara i andra språk. Första uttrycket är ett initialiseringsuttryck, som beräknas en gång. Andra uttrycket är ett avslutningsuttryck, som beräknas och testas före varje iterationssteg. Det tredje uttrycket är ett uttryck som beräknas efter varje iterationssteg. Normalt är första och sista uttrycken tilldelningsuttryck och det andra uttrycket ett relationsuttryck. Kommaoperatorn gör det dock möjligt att göra uttrycken omfattande. Hela beräkningsarbetet i en **for**-sats kan i vissa fall beskrivas med dessa tre uttryck. Ett eller flera uttryck kan också utelämnas:

```
for (;;) /* repetera i evighet */
  printf("Reaktionstest - slå Ctrl-C!\n");
```

Satser för att avbryta styrstrukturer och funktioner

Följande satser finns för att avbryta funktioner och olika slags satser:

break;	Avslutar närmaste omslutande **switch**, **while**, **do** eller **for**.
continue;	Avbryter pågående iterationssteg och påbörjar nästföljande i närmast omslutande **while**, **do** eller **for**.
return (i + 1);	Lämna funktion och returnera angivet uttrycks värde.
return;	Lämna funktion, returvärde odefinierat (”funktionen är en procedur”).
goto ERROR;	Ovillkorligt hopp till angivet läge, en identifierare. Räckvidden är begränsad till inom funktion.

Satserna **goto** och **continue** brukas vara villkorslöst förbjudna. Satsen **break** brukar tillåtas under vissa premisser, vanligtvis om läsbarheten ökas och/eller behovet av styrvariabler reduceras. Normalt vill man endast ha en utgång i underprogram, vilket innebär att endast ett **return** bör förekomma i en funktion. Flera **return** kan accepteras om läsbarheten ökas och/eller behovet av styrvariabler reduceras.

Funktioner

Funktion är den enda typen av underprogram i C, men kan användas som en procedur.

```
/*
 * Return sum of first n array elements.
 */
int sum(int a[], int n)
{
  int total = 0;
  while (n-- > 0)
    total += a[n];
  return total;
}

main()
{
  int  s, i, iarr[20];
  extern int sum(int, int);
  …
  s = sum(iarr, i);
  …
}
```

Externdeklarationen av funktionen `sum` i `main` är ej nödvändig om funktionerna finns i samma källkodsfil. Inte heller deklarationsordningen mellan funktionerna är av betydelse.

En funktion som ej returnerar något, dvs avses att anropas som en procedur, bör deklareras med resultattypen **void**. Normalvärdet är **int** om resultattyp ej anges.

I ANSI-standarden finns även ett äldre format för att definiera funktioner. Parametertyperna anges här efter själva funktionsspecifikationen. Funktionshuvudet för `sum` skrivs i detta format:

```
int sum (a, n)
int a[], n;
```

Detta förhindrar parametertypkontrollen för externa funktioner, eftersom parametertyperna ej anges i specifikationen:

```
main() {
  int  s, i, iarr[20];
  extern int sum();
  …
}
```

Det finns också i ANSI-standarden en s k ellipsnotation för funktioner som ska kunna ta ett variabelt antal parametrar. Exempelvis `printf` deklareras enligt denna som:

```
int printf(const char *format, …);
```

Stöd för att hantera variabla argumentlistor finns i `<stdarg.h>`.

Funktionsanrop

Det finns två vanliga sättatt anropa en funktion, antingen ”direkt” med funktionsnamn, eller genom en funktionspekare. Den första varianten har exempel visats på ovan. En funktionspekare är en pekarvariabel som innehåller adressen till en funktion:

```
{
  double  (*fp)(double);
  extern double  sin(double);
  fp = sin;
  (*fp)(0.5)
}
```

Parenteserna kring `*fp` är väsentliga eftersom annars `*fp` tolkas som ”funktion som returnerar pekare till **`double`**”, och ej ”pekare till funktion som returnerar **`double`**”. Adressen till standardfunktionen `sin` (inkludera <`math.h`>) tilldelas funktionspekaren `fp`. Ett funktionsnamn representerar alltid ett pekarvärde, adressen till funktionens kodsegment.

Funktionspekare används exempelvis för att överföra funktioner som parametrar till andra funktioner:

```
#include <math.h>
#define PI 3.1415;
main ()
{
  double  integral;
  integral = trapetz(0.0, PI, sin, 100);
  print("%f", integral);
}

/*
 *  Integralberäkning med trapetsmetoden
 *  för funktionen fp i [a,b] med steg n
 */
double trapetz (double a, double b, double (*fp)(double), int n)
{
  int     i;
  double  sum, h = (b - a) / n;
  sum = (*fp)(a) / 2 + (*fp)(b) / 2;
  for (i = 1; i < n; i++)
    sum += (*fp)(a + i * h);
  return (h * sum);
}
```

För anropet i `main`, kommer `trapetz` att i princip exekvera följande:

```
{
  int     i;
  double  sum, h = (b - a) / n;
  sum = sin(a) / 2 + sin(b) / 2;
  for (i = 1; i < n; i++)
    sum += sin(a + i * h);
  return (h * sum);
}
```

Notera att det är tillåtet att skriva:

```
sin;
```

dvs ett funktionsnamn utan att tilldela en funktionspekare eller att applicera funktionsoperatorn. Detta är dock helt meningslöst. Uttrycket `sin` ger som resultat det adressvärde som `sin` representerar. Det värdet tas ej om hand på något sätt, och satsen har följaktligen ingen effekt.

Man bör vara noga med att externdeklarera funktioner, även om detta ej är nödvändigt. Det går alldeles utmärkt att anropa exempelvis funktioner från standardbibliotek utan att inkludera dessa biblioteks inkluderingsfiler (t ex **`#include`** <`math.h`> i exemplet ovan). Det går också bra att anropa egendefinierade funktioner från annan källkodsfil utan att externdeklarera denna, eller att inkludera den filens inkluderingsfil. Det är bara det att då den fil som använder dessa externa funktioner kompileras, gör

kompilatorn vissa standardantaganden. Resultattypen antas vara **`int`**, och de formella parametrarna antas överensstämma med de aktuella, vilket kan ge helt felaktiga resultat om så ej är fallet.

Parameteröverföring

I C finns endast en parametermod realiserad och det är värdeanrop, *copy in*, dvs varje aktuell parameter beräknas och värdet överförs till funktionen. Detta betyder i princip att funktioner ej skulle kunna returnera värden via funktionsparametrar. Detta går dock att åstadkomma genom att överföra pekare till argumenten i stället för argumentens värden, och att funktioner deklarera motsvarande parametrar som pekare.Med operatorerna & och * kan således även referensanrop, *”call by reference”*, åstadkommas.

```
void swap (int  *i, int *j)
{
  int  old_i = *i;
  *i = *j;
  *j = old_i;
}
```

Funktionen `swap` anropas sedan enligt:

```
swap(&x, &y);
```

Observera att när det gäller parametrar som är av typ *fält* eller *funktion*, så representeras dessa *alltid* av pekare, så i dessa fall gäller i automatiskt alltid att en pekare till första tecknet i fältet respektive funktionskoden överförs till den formella parametern.

Det finns ingen garanti för beräkningsordningen för parametrar, och ej heller i vilken ordning värdena läggs på stacken. Det finns heller ingen exekveringskontroll av antal värden som överförs till en funktion, eller deras typ. Programmet lint kan dock hjälpa oss att kolla sådant. I standarden för C definieras hur en funktion kan anropas med varierande antal parametrar.

Huvudprogram

Varje C-programs exekvering startar i funktionen `main()`. Programargument kan kommas åt genom parametrarna `argv` och `argc`. Följande program skriver ut samtliga argument på kommandoraden som startade exekveringen av programmet:

```
/*
 * List arguments on command line.
 */
main (int argc, char **argv)
{
  register int  i;
  for (i = 0; i < argc; i++)
    printf("arg %i:%s\n", i, argv[i]);
}
```

Dynamisk minneshantering

Det finns ej någon mekanism i själva C för att hantera dynamiskt minne, men det finns en del standardfunktioner, och skulle man ej vilja använda dessa kan man själv relativt lätt realisera egen minneshantering. En standardfunktioner är `malloc`:

```
void *malloc (storlek)
```

som kan tilldela minne via en pekare. Pekartypen **`void`**`*`[1] anger att `malloc` kan returnera en pekare till vilken typ av objekt som helst. När `malloc` användes ska därför pekarvärdet som malloc returneras

1. I många äldre C-bibliotek returnerar `malloc`, liksom andra relaterade funktioner, `char*`.

omvandla till den typ av objekt vill tilldela minne för. Om minnestilldelning misslyckas, returnerar `malloc` värdet `NULL` (0). Om minne ska tilldelas för 100 heltal och kontroll om det gick bra görs, kan det se ut enligt följande:

```
int *ip;
…
if ( (ip = (int *) malloc(100 * sizeof(int))) == NULL) exit(1);
```

Minne kan ”återtilldelas” för att begära mer eller mindre minnesutrymme för ett dynamiskt objekt. Det kan ske med funktionen `realloc`:

```
ip = realloc(ip, 200);
```

det ”gamla” minnet återlämnas i princip genom detta och ”nytt” minne för 200 heltal tilldelas `ip`. Minne kan också återlämnas genom funktionen `free`:

```
free(ip);
```

Det finns fler standardfunktioner för dynamisk minnestilldelning definierade, flera av dem är variationer på ovan nämnda funktioner.

Preprocessorn

Då C-kompilatorn körs inleds bearbetningen av en preprocessor som utför operationer på vår källkod innan den går vidare till själva kompileringen. Ett # som första tecken på en rad anger att styrinformation till preprocessorn följer.

Symbolsubstitution

Symboler kan både definieras och ”avdefinieras”:

#**define** *identifierare sträng*

#**undef** *identifierare*

Används vanligtvis för att namnge litteraler, men kan i princip användas för att substituera godtyckliga texter:

```
#define SIZE  1000
#define EVER  (;;)
static int  stack[SIZE];

main ()
{
  for EVER {
    …
  }
}

#undef SIZE
```

Makron

Makron har parametrar och innebär litet mer avancerad bearbetning, speciellt som det är möjligt att använda tidigare definierade makron i en annan makrodefinition. Makron har formen:

#**define** *identifierare*(*identifierare*,…) *sträng*

Absolutbeloppsberäkning som ett makro:

```
#define ABS(A)  (((A) > 0) ? (A) : (-A))
```


Full parentetrisering krävs för att tillåta uttryck som argument och för att riskfritt kunna expandera makron i godtycklig omgivning. I annat fall kan exempelvis operatorprioriteter spela oss spratt. Används enligt:

```
i = ABS(i);
```

Användning av ett makro ser ut som ett funktionsanrop. Skillnaden är dock väsentlig. Ett funktionsanrop leder till att exekveringen tillfälligt fortsätter i funktionens kodkropp och att en del administration, som stackhantering med parameteröverföring etc sker. Ett makro däremot ersätts textuellt med makrotexten, där makrots parametrar substituerats. Ett makro ger därför vanligtvis en effektivare exekvering än en motsvarande funktion.

En del av standardbibliotekens ”funktioner” är egentligen makron, och en del operationer finns både som funktion och som makro.

Inkludering av filer

Det finns två former av inkluderingsdirektiv:

#**include** <*filnamn*>

#**include** "*filnamn*"

Preprocessordirektiv av detta slag ersätts av den namngivna filens innehåll. Första varianten anger att filen finns i ett standardbibliotek, `tex` `/usr/include` i UNIX. Den senare varianten anger att inkluderingsfilen skall sökas i samma filkatalog som den aktuella filen i första hand, fortsatt sökning görs i en till preprocessorn specificerad söklista.

Inkluderingsfilers namn slutar vanligtvis med `.h,` h som i *header file*.

Villkorlig kompilering

Följande direktiv tillåter att preprocessorn kan välja mellan olika kodavsnitt, när den producerar det program som så småningom ska kompileras. Det ger oss möjlighet att skriva kod för olika maskintyper, vilket ofta är nödvändigt, dels därför att C används i maskinnära tillämpningar, dels därför att en del konstruktioner i språket per definition är maskinberoende. En variant med tillåter ett generellt testvillkor:

```
#if VERSION == 1
  #include "version1.h"
#elsif  VERSION == 2
  #include "version2.h"
#else
  #include "standard.h"
#endif
```

En av tre förekommande inkluderingsfiler ersätter konstruktionen i kompileringsfilen. En annan variant testar om ett makro är definierat:

```
#ifdef BERKELEY
  p = vfork();
#else
  p = fork();
#endif
```

Om det tidigare i programmet förekommit definitionen #**define** `BERKELEY`, ersätts konstruktionen av den första satsen, annars av den andra.

Det finns också en `#ifndef`-version som testar om ett makro *ej* är definierat:

```
#ifndef BERKELEY
  p = fork();
#else
  p = vfork();
#endif
```

Det finns ytterligare makron än ovan nämnda.

In- och utmatning

Ingen in- och utmatning är definierat i själva språket, utan tillhandahålls i form av fördefinierade bibliotek, bland annat `stdio` (standard I/O). Då funktioner från `stdio` används bör filen `stdio.h` inkluderas. Den innehåller bland annat definitionen av datatypen `FILE`, definierar parametrar som används i funktionsanrop m.m.

`stdin`	standard inmatningsfil (0)
`stdout`	standard utmatningsfil (1)
`stderr`	standard felmeddelandefil (2)
`NULL`	tompekaren (0)
`EOF`	filslut (-1)

och en del makron för att effektivisera in- och utmatning (makron är effektivare än funktioner). `Stdio` innehåller grundläggande operationer för filhantering, ”pipe”-hantering, filstatus, samt olika former av in- och utmatning (formatterad, sträng, tecken och block).

Litteraturlista

[1] *C: A Reference Manual*, Harbison/Steele, Prentice-Hall, 1984.

[2] *Programming in C*, Miller/Quilici, Wiley, 1986.

[3] *The C Programming Language*, Second edition, Kernighan/Ritchie, Prentice-Hall, 1988.

[4] *Vägen till C*, Ulf Bilting/Jan Skansholm, Studentlitteratur, 1987.