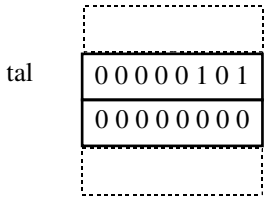
6 Lågnivåprogrammering

När språket C konstruerades hade man som en av målsättningarna att språket skulle kunna *hantera programmering på lågnivå*. Med lågnivå menas det som man tidigare behövt använda assemblerprogrammering till. Detta gäller exempelvis då man ska in och *manipulera enskilda bitar* i minnet.

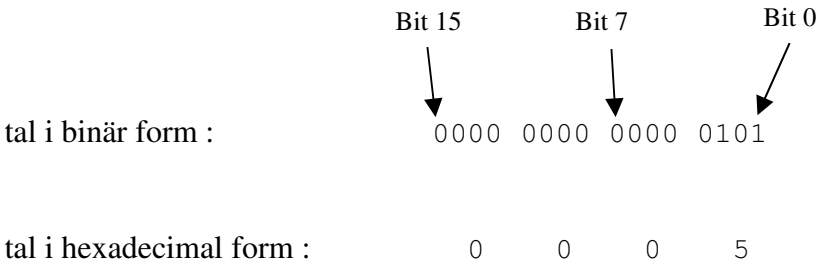
Normalt i ett högnivåspråk har man minnet åtkomligt byte-vis. Detta innebär att man kan läsa eller skriva hela byte i minnet. En byte är den minsta minnesdel som man normalt kan komma åt. Ibland, speciellt när man skriver program som ska arbeta mot någon typ av hårdvara, måste man exempelvis kunna kontrollera om en viss enskild bit är 0 eller 1. För att åstadkomma detta har man i C infört möjligheten att komma åt att *testa eller förändra* enskilda bitar i minnet.

Ex: Rita en minnesbild av variabeln tal samt sätt ut de enskilda bitarna då tal är definierad enligt:

```
short int tal = 5;
```



I vår implementation består tal av 2 byte eller 16 bitar. Man brukar rita bitarna med den *mest signifikanta biten,* bit nummer 15, först och den *minst signifikanta biten,* bit nummer 0, sist enligt:



Man kan komma åt enskilda bitar i C på i princip två olika sätt. Antingen kan man använda *bitoperatorer* eller också *bitfält*.

6.1 Bitoperatorer

Bitoperatorer kan verka på *alla typer av heltalsvariabler* och följande operatorer finns tillgängliga:

vänsterskift	<<	00001111 << 2	ger 00111100
högerskift	>>	00001111 >> 2	ger 00000011

OBS! Vänsterskift skiftar alltid in nollor från höger.
Högerskift skiftar alltid in nollor från vänster om *typen är unsigned*. För signed är det implementationsberoende vad som skiftas in.

Bitvis negation	~	~00001111	ger 11110000
Bitvis and	&	00001111 & 00001000	ger 00001000
Bitvis or	|	00001111 | 00001000	ger 00001111
Bitvis xor	^	00001111 ^ 00001000	ger 00000111

En typisk användning av bitoperatorerna är när man med hjälp av skift-operatorerna tillverkar en *bitmask,* som består av ettor i den eller de bitar som är aktuella och sedan använder någon av bitoperatorerna mellan operand och bitmask.

Ex: Antag att man vill *testa* om bit nummer 4 är satt eller inte i variabeln ch.

a) *Satt*

```
                          ch        0001 0100
mask = 1 << 4 ger         mask      0001 0000
res = ch & mask ger       res       0001 0000
```

b) *Ej satt*

```
                          ch        0000 0100
mask = 1 << 4 ger         mask      0001 0000
res = ch & mask ger       res       0000 0000
```

Efter detta kan man testa res logiskt om sant (!0) eller falskt (0).

Ex: Antag att man vill *sätta* bit nummer 7 i variabeln ch.

```
                          ch        0000 0100
mask = 1 << 7 ger         mask      1000 0000
ch = ch | mask ger        ch        1000 0100
```

Programmeringsmetodiskt ska man naturligtvis *tillverka färdiga funktioner* som man anropar för att testa eller sätta bitar. Man ska *undvika att hantera enskilda bitar* i onödan, eftersom det då *lätt slinker in fel* i programmet.

Ex: Skriv ett program som visar ASCII-koden för ett tecken i binär form, testar och sätter vissa bitar i tecknet som sedan visas igen.

```
#include <stdio.h>


void showbin(unsigned char c)
/* Visar c i binär form på skärmen */
{
   int bitnr;
   unsigned char mask;

   for (bitnr = 7; bitnr >= 0; bitnr--)
   {
      mask = 1 << bitnr;
      if (mask & c)
         putchar('1');
      else
         putchar('0');
   }
   putchar('\n');
}


int testbit(unsigned char c, int bitnr)
/* Testar om bit nummer bitnr satt */
{
   unsigned char mask = 1 << bitnr;

   if (c & mask)
      return 1;
   else
      return 0;
}


void setbit(unsigned char *c, int bitnr)
/* Sätter bit nummer bitnr */
{
   unsigned char mask = 1 << bitnr;

   *c = *c | mask;
}
```

```
void unsetbit(unsigned char *c, int bitnr)
/* Nollställer bit nummer bitnr */
{
   unsigned char mask = ~(1 << bitnr);

   *c = *c & mask;
}


void togglebit(unsigned char *c, int bitnr)
/* Togglar dvs byter satt blir ej satt och tvärtom */
{
   unsigned char mask = 1 << bitnr;

   *c = *c ^ mask;
}



int main()
{
   unsigned char ch = 0;

   printf(”Binär form : ”);
   showbin(ch);
   printf(”Sätt bit 5 : ”);
   setbit(&ch, 5);
   showbin(ch);
   printf(”Nollställ bit 5 : ”);
   unsetbit(&ch, 5);
   showbin(ch);
   printf(”Toggla bit 3 : ”);
   togglebit(&ch, 3);
   showbin(ch);
   if (testbit(ch, 3))
      printf("Bit 3 satt!\n");
   return 0;
}
```

En körning av programmet visar följande:

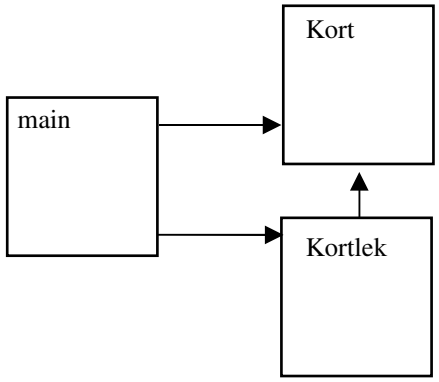
Binär form : 00000000
Sätt bit 5 : 00100000
Nollställ bit 5 : 00000000
Toggla bit 3 : 00001000
Bit 3 satt!

Det är inte bara när man ska testa eller sätta bitar, som man har nytta av att använda bitoperationerna. Har man ont om minne kan man *packa data* i mindre utrymmen och därmed spara på minne.

Ex: Skriv ett program som skriver ut två händer med 5 spelkort, bestående av färg mellan 1 och 4 och värde mellan 1 och 13, i varje hand. Korten ska utdelas från en blandad kortlek innehållande från början 52 kort i alla färger och valörer.
All data för ett kort ska rymmas i en byte med färgen 1 till 4 i bitarna 0 till 2 och valören 1 till 13 i resterande bitar enligt följande exempel, som visar ett kort med färgen 3 och valören 9 :

Valör	Färg
0 1 0 0 1	0 1 1

Då programmet ska hantera kort och kortlek delar man upp det i abstrakta datatyper enligt följande beroendediagram:



Principen att *spara* data i delar av en byte, som nedan har namnet kort, är:

färg = 3	färg	00000011
kort = färg	kort	00000011
valör = 9	valör	00001001
valör = valör << 3	valör	01001000
kort = kort | valör	kort	01001011

Principen att *hämta* data från en viss del av en byte, som heter kort, är :

	kort	01001011
färgmask = 7	färgmask	00000111
färg = kort & färgmask	färg	00000011 = 3
	kort	01001011
valörmask = 0xf8	valörmask	11111000
valör = kort & valörmask	valör	01001000
valör = valör >> 3	valör	00001001 = 9

```
/* Specifikation av kort - kort1.h */

#ifndef KORT1_H
#define KORT1_H

typedef unsigned char korttyp;

korttyp skapa_kort(unsigned char col, unsigned char val);
/* Returnerar ett kort med färgen col och valören val */

void visa_kort(korttyp kort);
/* Visar kortets färg och valör på skärmen */

#endif
```

```
/* Implementation av kort -- kort1.c */


#include <stdio.h>
#include "kort1.h"


korttyp skapa_kort(unsigned char col, unsigned char val)
{
   korttyp kort;

   kort = col;
   kort = kort | (val << 3);
   return kort;
}



void visa_kort(korttyp kort)
{
   unsigned char col, val;
   unsigned char colmask = 0x7;
   unsigned char valmask = 0xf8;

   col = kort & colmask;
   val = (kort & valmask) >> 3;
   printf("Färg : %d  Valör : %d\n", col, val);
}
```

```
/* Specifikation av kortlek -- kortlek.h*/

#include "kort1.h"

typedef
struct
{
   korttyp lek[52];
   unsigned char top;
} kortlektyp;

void skapa_kortlek(kortlektyp *klp);
/* Skapar kortlek med 52 kort */

void blanda_kortlek(kortlektyp *klp);
/* Blandar kortlek */

korttyp ge_kort(kortlektyp *klp);
/* Delar ut översta kortet */


/* Implementation av kortlek -- kortlek.c */

#include <stdlib.h>
#include <time.h>
#include "kortlek.h"

void skapa_kortlek(kortlektyp *klp)
{
   int i;

   for ( i = 0; i < 52; i++ )
      klp->lek[i] = skapa_kort(i/13 + 1, i%13 + 1);
   klp->top = 51;
}

void blanda_kortlek(kortlektyp *klp)
{
   int i, j, nr;
   korttyp temp;

   srand((unsigned)time(NULL));
   for (nr = 1; nr <= 1000; nr++)
   {
      i = rand() % 52;
      j = rand() % 52;
      temp = klp->lek[i];
      klp->lek[i] = klp->lek[j];
      klp->lek[j] = temp;
   }
}

korttyp ge_kort(kortlektyp *klp)
{
   return klp->lek[klp->top--];
}

/* Huvudprogram kortspel.c*/
```

```
#include <stdio.h>

#include "kort1.h"
#include "kortlek.h"

int main()
{
   kortlektyp kl;
   int i;

   skapa_kortlek(&kl);
   blanda_kortlek(&kl);
   printf("\nHand1\n");
   for (i = 1; i <= 5; i++)
      visa_kort(ge_kort(&kl));
   printf("\nHand2\n");
   for (i = 1; i <= 5; i++)
      visa_kort(ge_kort(&kl));
   return 0;
}
```

En körning av programmet kommer att slumpa fram två händer med 5 kort i varje hand som på skärmen visas enligt :

Hand 1
Färg : 2 Valör : 10
Färg : 1 Valör : 5
Färg : 1 Valör : 6
Färg : 3 Valör : 13
Färg : 4 Valör : 1

Hand 2
Färg : 3 Valör : 10
Färg : 1 Valör : 8
Färg : 2 Valör : 6
Färg : 2 Valör : 13
Färg : 3 Valör : 1

Tittar man på storleken av en kortlek så tar den upp 53 byte i minnet. 52 byte går åt till korten medan 1 byte går åt till top som håller reda på vilket kort som ska utdelas nästa gång.

6.2 Bitfält

Istället för att använda bitoperatorer finns det i C ett alternativt sätt att komma åt enskilda bitar i minnet, nämligen med hjälp av bitfält. Bitfält definierar man som en vanlig post, där man *med termerna i posten ger namn åt enskilda bitar eller grupper av bitar.*

Ex : Definiera ett bitfält, bf, så att man kan komma åt alla 8 enskilda bitar i en byte i minnet samt sätt bit nummer 4 och kontrollera om bit nummer 6 är satt.

```
struct byte
{
   unsigned int bit0 : 1;
   unsigned int bit1 : 1;
   unsigned int bit2 : 1;
   unsigned int bit3 : 1;
   unsigned int bit4 : 1;
   unsigned int bit5 : 1;
   unsigned int bit6 : 1;
   unsigned int bit7 : 1;
} bf;

bf.bit4 = 1;
if (bf.bit6)
   printf (”Bit nr 6 satt!\n”);
```

Ex : Definiera ett bitfält talbf, så att man i en byte kan sätta in två heltal mellan 0 och 15. Det första heltalet ska sparas i de 4 minst signifikanta bitarna och det andra i de 4 mest signifikanta bitarna. Sätt sedan det första talet till 5 och det andra till 10 samt skriv ut summan. Kallas talen för first och sec ska det se ut enligt:

sec	first
1010	0101

```
struct byte
{
   unsigned int first : 4;
   unsigned int sec : 4;
} talbf;

talbf.first = 5;
talbf.sec = 10;

printf(”Summan = %d\n”, talbf.first + talbf.sec);

/* Summan = 15, skrivs ut */
```

Bitfält kan ses *som ett alternativ* till att använda bitoperaratorer. Man kan generellt säga att bitfält är ett *säkrare* alternativ, eftersom man utnyttjar namnsättning av bitarna. På grund av detta blir det *också lättare att underhålla och införa förändringar* i programmen.

Ex: Skriv ett program som visar ASCII-koden för ett tecken i binär form, testar och sätter vissa bitar i tecknet, som sedan visas igen.

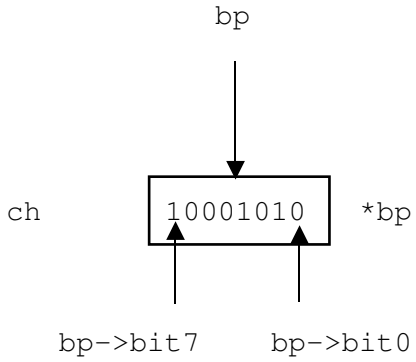
```
#include <stdio.h>

struct byte
{
   unsigned int bit0 : 1;
   unsigned int bit1 : 1;
   unsigned int bit2 : 1;
   unsigned int bit3 : 1;
   unsigned int bit4 : 1;
   unsigned int bit5 : 1;
   unsigned int bit6 : 1;
   unsigned int bit7 : 1;
};

void showbin(struct byte c)
{
   putchar(c.bit7 ? '1':'0');
   putchar(c.bit6 ? '1':'0');
   putchar(c.bit5 ? '1':'0');
   putchar(c.bit4 ? '1':'0');
   putchar(c.bit3 ? '1':'0');
   putchar(c.bit2 ? '1':'0');
   putchar(c.bit1 ? '1':'0');
   putchar(c.bit0 ? '1':'0');
   putchar('\n');
}

int testbit(struct byte c, int bitnr)
{
   if (bitnr == 0)
      return (c.bit0 == 1);
   else if (bitnr == 1)
      return (c.bit1 == 1);
   else if (bitnr == 2)
      return (c.bit2 == 1);
   else if (bitnr == 3)
      return (c.bit3 == 1);
   else if (bitnr == 4)
      return (c.bit4 == 1);
   else if (bitnr == 5)
      return (c.bit5 == 1);
   else if (bitnr == 6)
      return (c.bit6 == 1);
   else if (bitnr == 7)
      return (c.bit7 == 1);
}
```

```
void setbit(struct byte *c, int bitnr)
{
   if (bitnr == 0)
      c->bit0 = 1;
   else if (bitnr == 1)
      c->bit1 = 1;
   else if (bitnr == 2)
      c->bit2 = 1;
   else if (bitnr == 3)
      c->bit3 = 1;
   else if (bitnr == 4)
      c->bit4 = 1;
   else if (bitnr == 5)
      c->bit5 = 1;
   else if (bitnr == 6)
      c->bit6 = 1;
   else if (bitnr == 7)
      c->bit7 = 1;
}
void unsetbit(struct byte *c, int bitnr)
{
  if (bitnr == 0)
      c->bit0 = 0;
   else if (bitnr == 1)
      c->bit1 = 0;
   else if (bitnr == 2)
      c->bit2 = 0;
   else if (bitnr == 3)
      c->bit3 = 0;
   else if (bitnr == 4)
      c->bit4 = 0;
   else if (bitnr == 5)
      c->bit5 = 0;
   else if (bitnr == 6)
      c->bit6 = 0;
   else if (bitnr == 7)
      c->bit7 = 0;
}
void togglebit(struct byte *c, int bitnr)
{
   if (bitnr == 0)
      c->bit0 = c->bit0 ? 0 : 1;
   else if (bitnr == 1)
      c->bit1 = c->bit1 ? 0 : 1;
   else if (bitnr == 2)
      c->bit2 = c->bit2 ? 0 : 1;
   else if (bitnr == 3)
      c->bit3 = c->bit3 ? 0 : 1;
   else if (bitnr == 4)
      c->bit4 = c->bit4 ? 0 : 1;
   else if (bitnr == 5)
      c->bit5 = c->bit5 ? 0 : 1;
   else if (bitnr == 6)
      c->bit6 = c->bit6 ? 0 : 1;
   else if (bitnr == 7)
      c->bit7 = c->bit7 ? 0 : 1;
}
```

I huvudprogrammet kan man på vanligt sätt initiera bitfältet som en struct enligt:

```
struct byte b = {0, 1, 0, 1, 0, 0, 0, 1};
```

Man kan också använda en annan metod, där man skapar en pekare till bitfältet och tilldelar pekaren adressen för en tidigare definierad variabel. På detta sätt *namnger man bitarna i denna variabel*.




```
int main()
{
   unsigned char ch = 0x8a;
   struct byte *bp = (struct byte *)&ch;

   printf(”Binär form : ”);
   showbin(*bp);
   printf(”Sätt bit 5 : ”);
   setbit(bp, 5);
   showbin(*bp);
   printf(”Nollställ bit 5 : ”);
   unsetbit(bp, 5);
   showbin(*bp);
   printf(”Toggla bit 3 : ”);
   togglebit(bp, 3);
   showbin(*bp);
   if (testbit(*bp, 7))
      printf("Bit nr 7 satt!\n");
   return 0;
}
```

Resultatet av en körning blir :

Binär form : 10001010
Sätt bit 5 : 10101010
Nollställ bit 5 : 10001010
Toggla bit 3 : 10000010
Bit nr 7 satt!

Ex: Implementera den abstrakta datatypen kort enligt ovan med hjälp av ett bitfält innehållande färg och valör.

```
/* Specifikation av kort -- kort2.h */

#ifndef KORT2H
#define KORT2H

typedef
struct
{
   unsigned int color : 3;
   unsigned int value : 5;
} korttyp;

korttyp skapa_kort(unsigned char col, unsigned char val);
/* Returnerar ett kort med färgen col och valören val */

void visa_kort(korttyp kort);
/* Visar färg och valör av ett kort på skärmen */

#endif
```

```
/* Implementation av kort -- kort2.c */


#include <stdio.h>
#include "kort2.h"

korttyp skapa_kort(unsigned char col, unsigned char val)
{
   korttyp kort;

   kort.color = col;
   kort.value = val;
   return kort;
}


void visa_kort(korttyp kort)
{
   printf("Färg : %d  Värde : %d\n", kort.color, kort.value);
}
```

OBS! Programmet fungerar i övrigt med samma kortlek och kortspel som ovan. Man kan *ändra på implementationen av en abstrakt datatyp*, bara användargränssnittet i form av typnamn och funktionsdeklarationer är oförändrat.

6.3 Hårdvarunära programmering

När man ska skriva program som *ska ha kontakt med hårdvara* som bussar, printrar, I/O kort, bildskärmar etc, har man ofta behov av att *hantera enskilda bitar* i minnet. Detta beror på att hårdvarans register, som statusregister, dataregister etc ofta är *mappade till en adress* i minnet. Detta innebär att testa en speciell bit på en speciell adress i minnet är detsamma som att testa hårdvarans registerbit.

Ex: Antag att portarna på en dator är mappade till speciella adresser. Vill man läsa eller skriva till en port blir då detsamma som att läsa eller skriva ifrån en speciell minnesadress. De är inte enbart data som är mappat utan även statusregister för porten finns i minnet. I statusregistret kan man exempelvis kontrollera när data har kommit in och när det kan läsas etc.

I datorn ligger den första serieporten på adressen 0x03f80000 med data i första byte och ett antal statusregister i de efterföljande byten. Status för att kontrollera mottagning och sändning finns exempelvis på adressen 0x03f80005 där bit 0 sätts om klart för mottagning och bit 5 sätts om klart för sändning.

Om man exempelvis vill sända tecknet A via första serieporten ser minnet och därmed portens register ut som:

03f80000	01000001	(ASCII-kod 65 för A)
……		
03f80005	00100000	(Klart för sändning)

Skriv ett program som först sänder och sedan tar emot tecken via ovanstående dators serieport.

a) Genom att använda vanliga bitoperatorer

```
volatile unsigned char *dp = (unsigned char *)0x03f80000;

volatile unsigned char *sp = (unsigned char *)0x03f80005;


void skriv_port(char ch)
{
   *dp = ch;                        /* Skriv */

   while (! (*sp & (1 << 5)))       /* Vänta */
      ;
}


char las_port(void)
{
   while (! (*sp & 1 ))             /* Vänta */
      ;
   return *dp;                      /* Läs */
}



void main()
{
   char ch;

   /* Läs värde från port till ch */
   ch = las_port();

   /* Skriv värdet av ch till port */
   skriv_port(ch);

}
```

OBS! Med volatile markeras att kompilatorn ej får optimera bort det angivna minnesutrymmet till ett register. Man *säkerställer därmed att programmet alltid arbetar mot angiven adress*.

b) Genom att använda bitfält

```
volatile unsigned char *dp = (unsigned char *)0x03f80000;


struct status
{
   unsigned int rxready :1;
   unsigned int overrun :1;
   unsigned int parerr  :1;
   unsigned int framerr :1;
   unsigned int rxbreak :1;
   unsigned int txready :1;
   unsigned int txfifo  :1;
                        :1;
} *sp = (struct status *)0x03f80005;



void skriv_port(char ch)
{
   *dp = ch;

   while (!sp->txready)
      ;
}


char las_port(void)
{
   while (!sp->rxready)
      ;

   return *dp;
}



int main()
{
   char ch;

   /* Läs värde från port till ch */
   ch = las_port();

   /* Skriv värdet av ch till port */
   skriv_port(ch);

   return 0;
}
```