Typer och lagringsklasser

Ur innehållet

- Variabler och minnesanvändning
- Mer typer (union/uppräkningstyp/bitfält)

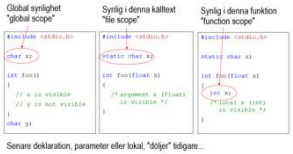
Målsättningar:

- Kunna använda adressering av portar med struct/union/bitfält
- Kunna dela upp och strukturera ett större programs källtexter

Variabler och minnesanvändning

Ett deklarerat objekt karakteriseras av:

- Synlighet och åtkomlighet



- Varaktighet: permanent plats i minnet eller tillfällig (register/stack)
- Typ av bindning: ingen, intern eller extern

Permanent lagring

Variabler med permanent adress i minnet:

```
int i; /* global synlighet */

static int i; /* synlig i denna källtext */

void func(void) {
  static int i; /* synlig i funktionen sub */
}
```

I ARM/Thumb assemblerspråk:

```
            .GLOBL    i
i:          .SPACE    4
.i:         .SPACE    4
.func_i:    .SPACE    4
```

alternativt sätt för global variabel:

```
            .comm gi,4,4
(.comm sym,length,alignment)
```

Symboler med begränsad synlighet tilldelas internt *unika* namn av kompilatorn. Kompilatorgenererade namn dyker ofta upp med inledande '.' i assemblerkälltexten. Samma C-symboler kan därför användas i olika sammanhang utan konflikt.

Tillfällig lagring – i register eller på stacken

Exempel:
Gör en lämplig registerallokering och koda tilldelningssatserna i funktionen:

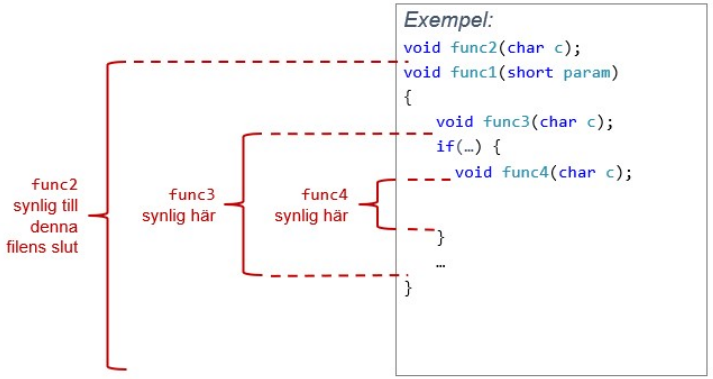
```
void   f (int a, int b, int c )
{
    int d:
    int e;
    d = a;
    e = b;
    …
}
```

Lösning:
Vi ser att R0,R1 och R2 redan är upptagna av parametrar, R3 upplåter vi för uttrycksevaluering i funktionen och gör registerallokeringen:
`d=R4, e=R5.`
Vi får då följande kodning:

```
@void   f (int a, int b, int c )
f:
@{
@    int d:
@    int e;
     PUSH    {R4,R5,LR}
     MOV     R4,R0
     MOV     R5,R1
     …
     POP     {R4,R5,PC}
@}
```

Synlighet

Synligheten kan begränsas av deklarationens block.



Varaktighet ("storage duration")

Varje deklarerat objekt har en *varaktighet* under vilken det kan användas.
Det finns fyra olika typer av varaktighet i C:

- **automatic**: reserverat minnesutrymme finns i det block objektet deklarerats. ("tillfällig lagring")
- **static**: reserverat minnesutrymme under hela programexekveringen, initieras innan exekvering av main-funktionen.("permanent lagring")
- **allocated**: minnesutrymme som reserverats med funktioner för dynamisk minneshantering (`malloc/free`).
- **thread**: reserverat minnesutrymme under hela exekveringen av den tråd där objektet deklarerats. Jämför med "static" men inte hela programmet.

Bindning – ("linkage")

Med "bindning" menas hur en variabel (eller funktion) kan refereras:

Ingen bindning, variabeln kan refereras enbart inom dess åtkomlighet.

Extern bindning, variabeln kan refereras från externa delar (samtliga källfiler) i programmet.

Intern bindning, variabeln kan refereras från den källfil (translation unit) den deklarerats.

Lagringsklasser ("storage class")

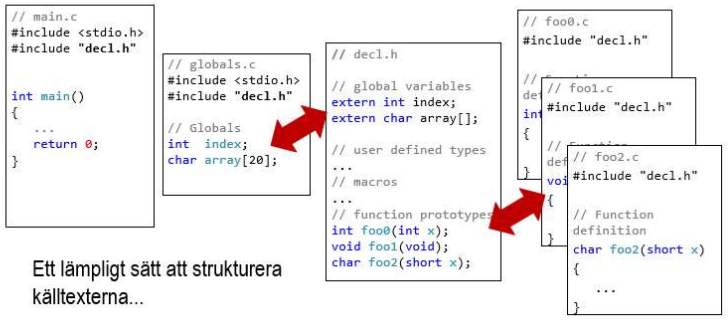
Med lagringsklassen specificeras objektets varaktighet och bindning
En lagringsklass kan anges med ett *tillägg* i deklarationen:

`auto` – tillfällig lagring, ingen bindning
`register` – tillfällig lagring, ingen bindning; adressoperator kan inte användas
`static` – permanent lagring intern bindning (dock ej om deklaration i block)
`extern` – permanet lagring och extern bindning (om inte redan deklarerad)

Exempel:

```
static int var1 = 1;
extern int var2;
void func()
{
   auto int var3 = 0;  /* auto är default här */
   register auto int var4 = 0; /* lämplig för registerallokering*/
}
```

Organisation av applikationsprogrammet



Ett lämpligt sätt att strukturera källtexterna...

Typ union

`union` tillåter oss att lagra olika datatyper på samma plats i minnet.
Syntaxen är den samma som för `struct`

```
union opt_name {
      int a;
      char b;
      float c;
 } x;
 x.a = 4;
 x.b = 'i';
 x.c = 3.0;

typedef union {
  float v[2];
  struct { float x,y; };
} Vec2f;

Vec2f pos;
pos.v[0] = 1; pos.v[1] = 2;
pos.x = 1; pos.y = 2;
```

```
&x.a == &x.b == &x.c
a, b och c delar minnesadress. Samma address kan alltså
refereras på tre olika sätt, via tre olika variabelnamn.
```

```
pos.v[0] och pos.x är de samma. "pos.x" säger tydligt att vi
menar the x-koordinaten. "pos.v[i]" är användbar om vi vill
"loopa" över alla x- och y- koordinater.

Exempel:
Vec2f addVec(Vec2f a, Vec2f b)
{
    for(i=0; i<2; i++)
        a.v[i] += b.v[i];
   return a;
}
```

Bitfältstyp, ("bitfield")

Ett bitfält är en struct- (eller union-) medlem av heltalstyp:
`identifierare(kan utelämnas): width.`

- `width` är en heltalstyp (`char`/`short`/`int`/`long..`) >= 0
  - `width>0:` anger antalet bitar i fältet
  - `width==0:` anger att nästa bitfält ska starta på en gräns för heltalstypen
- Det finns ingen pekartyp för bitfält
- `sizeof`-operator kan inte användas med bitfält

Exempel:

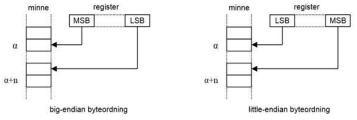
```
struct S {
 unsigned int b1 : 5;
 unsigned int :0; // börja med ny unsigned i
 unsigned int b2 : 6;
 unsigned int b3 : 15;
};
```

```
// 5 bitar: värdet av b1
// 27 bitar: används ej
// 6 bitar: värdet av b2
// 15 bitar: värdet av b3
// 11 bits: används ej
```

Uppräkningstyp, `enum`

```
enum type_name { value1, value2,..., valueN };
//type_name kan utelämnas. Default: value1 = 0, value2 = value1 + 1, etc.

enum type_name { value1 = 0, value2, value3 = 0, value4 };
//Man kan sätta startvärden, with gcc values: 0, 1, 0, 1.

enum day {monday=1, tuesday, wednesday, thursday, friday, saturday, sunday};
enum day today; // day kan vara char, short eller int
today=wednesday; printf("%d:th day",today+1); // output: "4:th day"

typedef enum { false, true } bool;
bool ok = true;
-------------------------------------------------------------------------------
#define B_E       0x40
#define B_RST     0x20
#define B_CS2     0x10
#define B_CS1     8
#define B_SELECT  4
#define B_RW      2
#define B_RS      1

Kan ersättas med:
enum {B_RS=1, B_RW=2, B_SELECT=4, B_CS1=8, B_CS2=0x10, B_RST=0x20, B_E=0x40};
```

Byte- och bitordning "endianess"



I vårt laborationssystem, ST32F407, använder vi little-endian.

Byte-adressering med union

```
// GPIO
typedef unsigned int uint32_t;
typedef unsigned char uint8_t;

typedef volatile struct tag_gpio {
    uint32_t    moder;
    uint32_t    otyper;
    uint32_t    ospeedr;
    uint32_t    pupdr;
    union {
        uint32_t    idr;
        struct {
            uint8_t idrLow;
            uint8_t idrHigh;
            short   reserved;
        };
    };
    union {
        uint32_t    odr;
        struct {
            uint8_t odrLow;
            uint8_t odrHigh;
            short   reserved;
        };
    };
} GPIO;
#define GPIO_D (*((volatile GPIO*) 0x40020c00))
#define GPIO_E (*((volatile GPIO*) 0x40021000))
```

Exempel:
Nu kan idrHigh adresseras:

```
uint8_t  c = GPIO_E.idrHigh;
```

Snarare är :

```
uint8_t  c = *((uint8_t *)&(GPIO_E.idr) + 1));
```

Exempel:

```
GPIO_E.odrLow &= ( ~B_SELECT & ~x);
```

i stället för:

```
*((uint8_t *)&(GPIO_E.odr)) &=(~B_SELECT & ~x);
```

Adressering med bitfält

```
// GPIO
typedef volatile struct tag_gpio {
  ...
  union {
    uint32_t    idr;
    struct {
      uint8_t   idrLow;
      union{
        uint8_t idrHigh;
        uint8_t col:4;
      };
      short     reserved;
    };
  };
  union {
    uint32_t    odr;
    struct {
      uint8_t   odrLow;
      union{
        uint8_t odrHigh;
        uint8_t unused:4,row:4;
      };
      short     reserved;
    };
  ...
} GPIO;
```

```
I stället för...
void kbdActivate( unsigned int row )
{
      switch( row )
      {
            case  1: *GPIO_D_ODRHIGH = 0x10 ; break;
            case  2: *GPIO_D_ODRHIGH = 0x20 ; break;
            case  3: *GPIO_D_ODRHIGH = 0x40 ; break;
            case  4: *GPIO_D_ODRHIGH = 0x80 ; break;
            case  0: *GPIO_D_ODRHIGH = 0x00;  break;
      }
}
```

```
... kan vi då koda:
void kbdActivate( unsigned int row )
{
      switch( row )
      {
            case  1: GPIO_D.col = 1 ; break;
            case  2: GPIO_D.col = 2 ; break;
            case  3: GPIO_D.col = 4 ; break;
            case  4: GPIO_D.col = 8 ; break;
            case  0: GPIO_D.col = 0;  break;
      }
}
```