

# Sammanfatta datatyper

## Ur innehållet

- Användardefinierade typer, "typedef"
- En komplex, användardefinierad typ "struct" ("post")
- Portbeskrivning med "struct"
- Inkapsling av data och funktioner
- Mer typer (union/uppräkningsstyp/bitfält)

## Läsanvisningar:

Lärobok kap 5.4

## Målsättningar:

- Kunna använda sammansatta typer för inkapsling ("objekt")
- Kunna använda adressering av portar med struct/union/bitfält

# typedef - alias för en typ

`typedef` används för att skapa ett *alias*, oftast för att förenkla och förkorta typuttryck. Avsikten är att förtydliga och öka läsbarheten, syntaxen är:

`typedef typ alias_typnamn [, alias_typnamn ]... ;`

Exempel:

```
typedef unsigned char    uint8;
typedef short int        int16;
// Anm. '*' är en del av alias-namnet och ingår INTE i befintlig typ
typedef unsigned char    *ucharptr;

                        typ          alias_typnamn

uint8 a, b = 0, c;
ucharptr p;
typedef int postnr;
typedef int strtNr;
postnr x = 41501;
strtNr y = 3;
x = y; // Typriktigt, helt ok
```

# struct ("post"), en sammansatt datatyp

- Har en eller flera medlemmar (*fields*).
- En medlem kan vara av godtycklig typ, exempelvis:  
`int`, `char`, `long` (`signed/unsigned`), `float`, `double`
- Användardefinierad (med "`typedef`")
- Sammansatt typ (dvs. en annan `struct`).
- Alla typer av pekare

Deklarationssyntax:

```
struct structnamn{  
    medlem;  
    [medlem; [medlem;]... ]  
};
```

# Deklaration av "struct" (post)

```
// Deklaration av structens typ
struct structnamn{
    ...
};

// Deklaration av structens typ och
// en variabel av denna typ
struct structnamn{
    ...
} variable;

// Det är vanligt med användardefinierade struct-typer
typedef struct structnamn TSTRUCTNAMN;
// alternativt:
typedef struct structnamn /* structnamn kan utelämnas här */
{
    ...
} TSTRUCTNAMN;
```

*Exempel:*

```
// Datatyp för koordinater
typedef struct coord
{
    int x,y;
} COORD;
// VariabeldeklARATIONER
struct coord start,end;
//eller..
COORD start, end;
```

# Användning

```
#include <stdio.h>

typedef struct coord{ int x,y; } COORD;

int main()
{
    struct coord start;
    COORD end;

    start.x = 10; start.y = 10;
    end.x   = 20; end.y   = 20;

    printf("Line starts at (x,y) %d,%d \n
           and ends at (x,y) %d,%d \n",
           start.x, start.x,
           end.x, end.x);

    return 0;
}
```

Typen kan användas på två sätt:

`struct coord` eller  
`COORD`

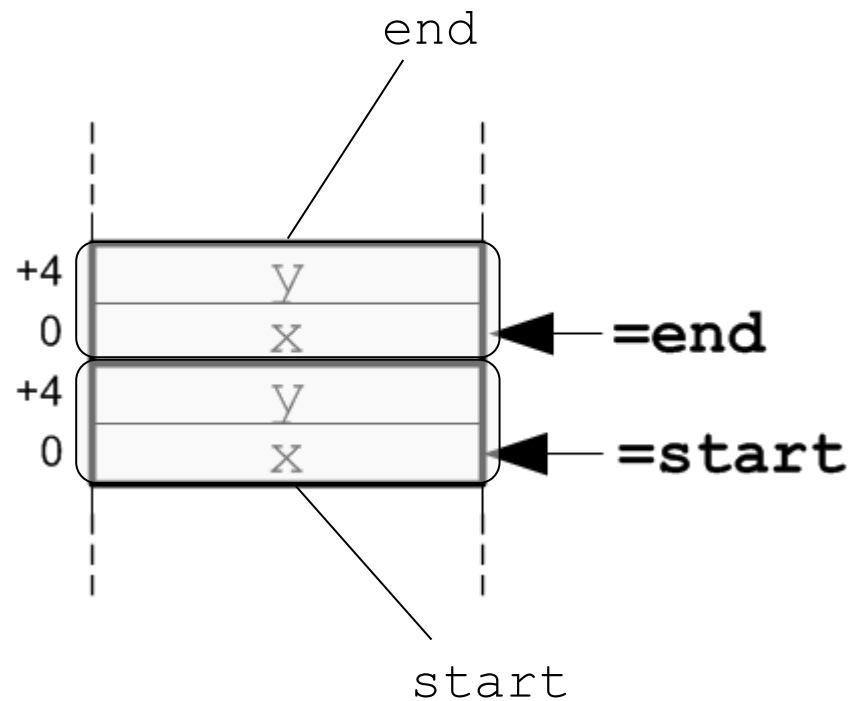
Likvärdiga deklarationer av  
variablerna `start` och `end`

Medlemmarna refereras via `"."`-  
operatören ("punktoperatören")

# Referenser

*Exempel:* Datatyp för koordinater

```
typedef struct
{
    int x,y;
} COORD;
```



*Exempel:* Vi har deklarationerna:

```
COORD start, end;
```

Koda tilldelningen:

```
start.y = end.y;
```

i ARM/Thumb assemblerspråk

*Lösning:*

```
LDR R0, =end      @ R0 ← &end
LDR R0, [R0, #4]  @ R0 ← end.y
LDR R1, =start    @ R1 ← &start
STR R0, [R1, #4]  @ start.y = end.y
```

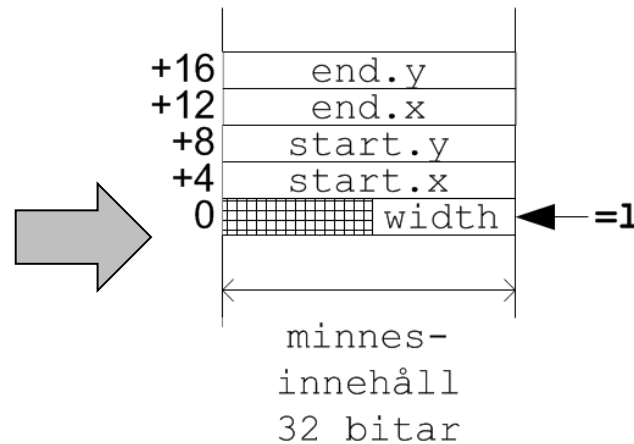
# Inbäddad post

*Exempel:* Datatyp för koordinater  
`typedef struct`  
{  
    `int x,y;`  
} `COORD;`

Vi har deklarationen:

```
typedef struct {  
    short width;  
    COORD start;  
    COORD end;  
} LINE, *PLINE;  
LINE l;
```

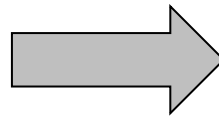
*Exempel:*



```
.ALIGN    2  
offset_width = 0  
offset_start.x = offset_width + sizeof(width) + align(2)  
offset_start.y = offset_start.x + sizeof(start.x) + align(2)  
offset_end.x = offset_start.y + sizeof(start.y) + align(2)  
offset_end.y = offset_end.x + sizeof(end.x) + align(2)
```

Visa kodsekvenser som  
evaluerar följande uttryck i  
register R0.

- a) `l.width`
- b) `l.end.y`

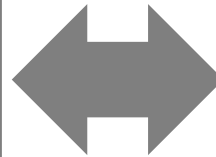


```
a)  
LDR    R0, =1  
LDRH   R0, [R0, #0]      @ R0= 1+ 0 (offset_width)  
b)  
LDR    R0, =1  
LDR    R0, [R0, #16]     @ R0= 1+ 16 (offset_end.y)
```

# Alternativa deklarationer

```
struct line
{
    struct coordinate
    {
        int x,y;
    } start, end;
} linje;
// Referenser
linje.start.x;
linje.end.y; // etc
```

`struct coordinate`  
kan bara användas som medlem av  
`struct line`.



```
struct coordinate
{
    int x,y;
};

struct line
{
    struct coordinate start;
    struct coordinate end;
} linje;
// Referenser
linje.start.x;
linje.end.y; // etc
```

`struct coordinate`  
och  
`struct line`  
kan användas var och en för sig.

# Initiering, fullständig och ofullständig

En `struct` kan initieras med en lista.

Medlemmarna tilldelas värden i samma ordning som deklarationen.

Listan kan vara ofullständig, dvs. alla medlemmar behöver inte initieras.

```
struct Course {  
    char* name;  
    float credits;  
    int numberOfParticipants;  
};
```

```
struct Course c1 = {"MOP", 7.5, 110};
```

```
struct Course c2 = {"MOP", 7.5};
```

Fullständig initiering

Ofullständig initiering

# Ofullständig deklaration

En struct kan först ges som en ofullständig deklaration, för att senare definieras fullständigt.

*Exempel:*

```
struct coordinate; /* coordinate är en struct */
```

```
struct line
{
    struct coordinate start;
    struct coordinate end;
};
```

```
struct linepointers
{
    struct coordinate *start;
    struct coordinate *end;
};
```

*Fel:*

Storleken av `struct coordinate` är okänd, alltså kan inte storleken av `struct line` bestämmas.

*Ok:*

En pekarstorlek är alltid känd, alltså kan storleken av `struct linepointers` bestämmas.

# Pekare till **struct**, pilnotation

Det är vanligt att använda pekare och pilnotationen förenklar detta betydligt.

*Exempel:*

```
struct coordinate {int x,y;} c1;
struct coordinate *ptr;

ptr = &c1;
// Direkt, variabel via punktoperator
c1.x = ...
// eller via derefererad pekare
(*ptr).x = ...
// eller kortare, piloperatorn:
p->x    = ...
```

*Exempel:* Koordinater i en länkad lista

```
typedef struct tPoint{
    char x;
    char y;
    struct tPoint *next;
} POINTLIST;

/* Gå igenom lista av koordinater... */
POINTLIST *first; /* pekar på listan */
...
POINTLIST *p = first;
while( p != (POINTLIST *) 0 )
{
    /* Gör något med koordinaten ... */
    p = p->next;
}
```

# Pekare till struct, pilnotation

## Exempel:

```
typedef struct coord
{
    int x,y;
    struct coord *next;
} COORD;
COORD c1, ... , c10;
COORD *first, *p;
```

## Koda tilldelningarna:

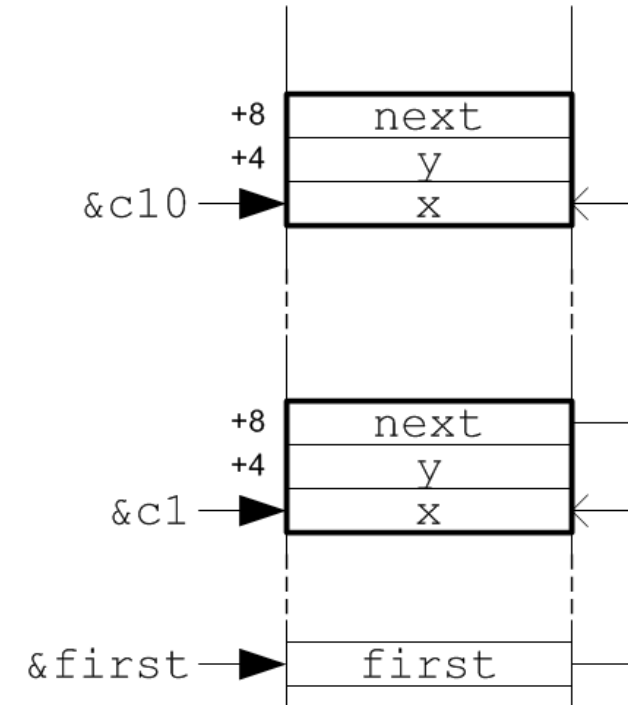
```
first = &c1;
p = first;
p->next = &c10;
```

`(*p).next = &c10;`  
`p->next = &c10;`

```
@ first = &c1;
    LDR    R0,=c1
    LDR    R1,=first
    STR    R0,[R1]
```

```
@ p = first;
    LDR    R0,=first
    LDR    R0,[R0]
    LDR    R1,=p
    STR    R0,[R1]
```

```
@ p->next;
    LDR    R0,=c10
    LDR    R1,=p
    LDR    R1,[R1]
    STR    R0,[R1,#8]
```



# Portadressering med poster

struct-typen är också användbar för att deklarerar portar

```
// Som alternativ till :
#define portModér ((volatile unsigned int *) (GPIO_BASE))
#define portOtyper ((volatile unsigned int *) (GPIO_BASE+0x4))
#define portOspeedr ((volatile unsigned int *) (GPIO_BASE+0x8))
osv.
```

// kan vi använda en post-definition som:

```
typedef volatile struct {
    unsigned int modér;
    unsigned int otyper; // +0x4
    unsigned int ospeedr; // +0x8
    unsigned int pupdr; // +0xC (12)
    unsigned int idr; // +0x10
    unsigned int odr; // +0x14
    unsigned int bsrr; // +0x18
    unsigned int lckr; // +0x1C
    unsigned int afrl; // +0x20
    unsigned int afrh; // +0x24
} GPIO, *PGPIO;
```

| offset | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Register     |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|--------------|
| 0      |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   | GPIO_MODER   |
| 4      |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   | GPIO_OTYPER  |
| 8      |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   | GPIO_OSPEEDR |
| 0xC    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   | GPIO_PUPDR   |
| 0x10   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   | GPIO_IDR     |
| 0x14   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   | GPIO_ODR     |
| 0x18   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   | GPIO_BSRR    |
| 0x1C   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   | GPIO_LCKR    |
| 0x20   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   | GPIO_AFRL    |
| 0x24   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   | GPIO_AFRH    |

Exempel:

```
#define GPIO_D (*(volatile PGPIO) 0x40020c00)
#define GPIO_E (*(volatile PGPIO) 0x40021000)
```

```
GPIO_E.modér = 0x55555555;
GPIO_E.otyper = 0x00000000;
GPIO_D.modér = 0x55550000;
GPIO_D.pupdr |= 0x00005555;
```

# Portadressering med poster

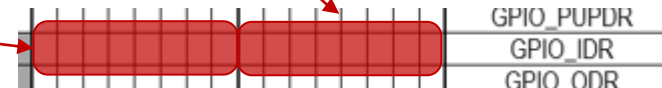
Deklarationen kan i stället anpassas till portens olika registers storlekar

```
typedef volatile struct {  
    unsigned int  moder;  
    unsigned int  otyper;  
    unsigned int  ospeedr;  
    unsigned int  pupdr;  
    unsigned int  idr;  
    unsigned int  odr;  
    unsigned int  bsrr;  
    unsigned int  lckr;  
    unsigned int  afrl;  
    unsigned int  afrh;  
} GPIO, *PGPIO;
```

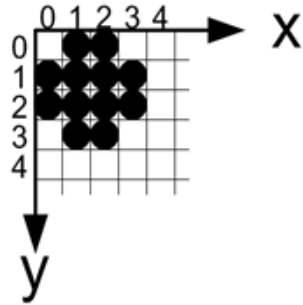
```
typedef volatile struct {  
    unsigned int      moder;  
    unsigned short int otyper;  
    unsigned short int Reserved0;  
    unsigned int      ospeedr;  
    unsigned int      pupdr;  
    unsigned char     idrLow;  
    unsigned char     idrHigh;  
    unsigned short int Reserved1;  
    unsigned char     odrLow;  
    unsigned char     odrHigh;  
    unsigned short int Reserved2;  
    unsigned int      bsrr;  
    unsigned short int lckr;  
    unsigned short int Reserved3;  
    unsigned int      afrl;  
    unsigned int      afrh;  
} GPIO, *PGPIO;
```

Konvertering till lämplig storlek:

```
GPIO GPIO_E;  
typedef unsigned char uint8_t;  
uint8_t x = *(uint8_t*)&GPIO_E.idr;  
uint8_t y = *((uint8_t*)&GPIO_E.idr+1);
```



# Inkapsling av data, ett "objekt"



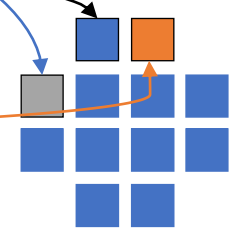
```
typedef struct tPoint{
    unsigned char x;
    unsigned char y;
} POINT;

#define MAX_POINTS 20

typedef struct tGeometry{
    int    numpoints;
    int    sizex;
    int    sizey;
    POINT  px[ MAX_POINTS ];
} GEOMETRY, *PGEOMETRY;
```

Exempel: Arbetsboken, avsnitt 5.4

```
// Skapa och initiera ett object av typen GEOMETRY:
GEOMETRY ball_geometry = {
    12, 4, 4,    // numpoints, sizex, sizey
    { // POINT px[20]
        {0,1},
        {0,2},
        {1,0},
        {1,1},
        {1,2},
        {1,3},
        {2,0},
        {2,1},
        {2,2},
        {2,3},
        {3,1},
        {3,2}    // Ofullständig initiering
    }           // (12 av 20)
};
```



# Poster med funktionspekare

Inkapsling av pekare till funktioner.

**Exempel:**

```
typedef struct tObj {  
    PGEOMETRY geo;  
    int dirx, diry;  
    int posx, posy;  
    void (*draw)(struct tObj *);  
    void (*clear)(struct tObj *);  
    void (*move)(struct tObj *);  
    void (*set_speed)(struct tObj *, int, int);  
} OBJECT, *POBJECT;
```

`draw`, `clear`, `move` och `set_speed`  
är pekarter.

Kan uppfattas som  
"metoder" i  
objektorienterat språk.

# Poster med funktionspekare

Man kan ge objektet dess initiala egenskaper med statisk initiering ....

*Exempel statisk initiering:*

```
OBJECT ball = {  
    &ball_geometry,  
    0, 0,  
    64, 32,  
    draw_object,  
    clear_object,  
    move_object,  
    set_object_speed  
};
```

```
typedef struct tObj {  
    PGEOMETRY geo;  
    int dirx, diry;  
    int posx, posy;  
    void (*draw)(struct tObj *);  
    void (*clear)(struct tObj *);  
    void (*move)(struct tObj *);  
    void (*set_speed)(struct tObj *, int, int);  
} OBJECT, *POBJECT;  
  
void move_object(POBJECT o ) {...}  
void draw_object(POBJECT o ) {...}  
void clear_object(POBJECT o ) {...}  
void set_object_speed(POBJECT o, int x, int y ) {...}
```

...medan dynamisk initiering sker under programmets gang.

*Exempel dynamisk initiering:*

```
OBJECT ball;  
ball.geo = &ball_geometry;  
ball.dirx = 0; ball.diry = 0;  
ball.posx = 64; ball.posy = 32;  
ball.draw = draw_object;  
ball.clear = clear_object;  
ball.move = move_object;  
ball.set_speed = set_object_speed;
```

# Poster med funktionspekare

Dynamisk initiering kan exempelvis användas för att ändra egenskaper hos ett objekt baserat på olika händelser.

```
typedef struct tObj {
    PGEOMETRY geo;
    int dirx, diry;
    int posx, posy;
    void (*draw)(struct tObj *);
    void (*clear)(struct tObj *);
    void (*move)(struct tObj *);
    void (*set_speed)(struct tObj *, int, int);
} OBJECT, *POBJECT;

void move_object(POBJECT o) {...}
void draw_object(POBJECT o) {...}
void clear_object(POBJECT o) {...}
void set_object_speed(POBJECT o, int x, int y) {...}
```

*Exempel: alternativ "move"-funktion:*

```
// Vi definierar en alternativ move-funktion
void alternate_move_object(POBJECT o) { ... }
```

```
POBJECT pb;
pb = &ball;
pb->move = move_object;
...
pb->move( pb ); // Här utförs move_object
...
pb->move = alternate_move_object;
...
pb->move( pb ); // Här utförs alternate_move_object
...
```

# Objektorientering i ett icke-objektorienterat språk

- Ideer kring inkapsling/objektorientering har utvecklats sedan 1950-talet
- **C++** är ett objektorienterat språk, inkapsling finns exempelvis som konstruktioner i språket.
- Lokala funktioner tillsammans med data, konceptet kallas "klass" - en klass kan innehålla både variabler och funktioner.
- C har inte klasser, men vi kan simulera en klass med hjälp av sammansatta typer.

## Inkapsling: klass i C++

```
typedef struct {  
    int a;          // a member  
    void inc() {    // method that increments a  
        a++;  
    }  
} MyClass;
```

```
MyClass v = {0}; // initialize variable  
v.inc();         // increment variable
```

Inte i C...

...men vi kan åstadkomma samma sak:

```
typedef struct tMyClass{  
    int a;  
    void (*inc) (struct tMyClass* this);  
} MyClass;
```

```
void incr(MyClass* this)  
{  
    this->a++;  
}
```

```
MyClass v = {0, incr};  
v.inc(&v);
```

Observera!

# Objektorientering med C

Detta...

```
typedef struct tMyClass {  
    int    a;  
    void (*inc) (struct tMyClass* this);  
} MyClass;
```

`tMyClass` används för att ange parameterns typ eftersom `MyClass` fortfarande är odefinierad.

... är ekvivalent med detta:

```
struct MyClass;  
typedef struct {  
    int    a;  
    void (*inc) (MyClass* this);  
} MyClass;
```

Ofullständig deklaration av `MyClass`, den fullständiga deklarationen ges av den sista raden...

..men en pekare behövs här

Båda metoderna fungerar...

Hur metodanropet `.inc(...)` fungerar i C:

```
// MyClass kallar vi objekt.  
// v1, v2 är två instanser av objektet.  
MyClass v1 = {0, incr}, v2 = {1, incr};  
// Metodanrop inc() för v1 och v2.  
v1.inc(&v1);  
v2.inc(&v2);
```

```
// incr() är en vanlig function  
// som både v1.inc och v2.inc pekar på.  
void incr(MyClass* this)  
{  
    this->a++;  
}
```

Initieringarna gör att `v1.a = 0`, `v2.a = 1`, `v1.inc` och `v2.inc` pekar på funktionen `incr()`. Minns att `incr` bara är en symbol för den minnesadress där funktionen `incr()` är placerad och `v1.inc` och `v2.inc` är pekarvariabler som nu innehåller denna adress.

Här har vi anropen av de funktioner som pekarvariablerna anger med `&v1` respektive `&v2` som parametrar. Så `v1.inc(&v1)` är nu samma sak som anropet `incr(&v1)` och `v2.inc(&v2)` är samma sak som anropet `incr(&v2)`.

Anropet `incr(&v1)` betyder att parametern `this` utgör adressen till `v1`. Alltså har vi att `this->a++` är samma som `(&v1)->a++` (vilket är ekvivalent med `v1.a++`)..... vilket är precis det vi vill göra med `v1.inc(&v1)`; Det samma gäller `v2.inc(&v2)`; Dvs. inkrementering av `v2.a`.

# 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 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;  
}
```

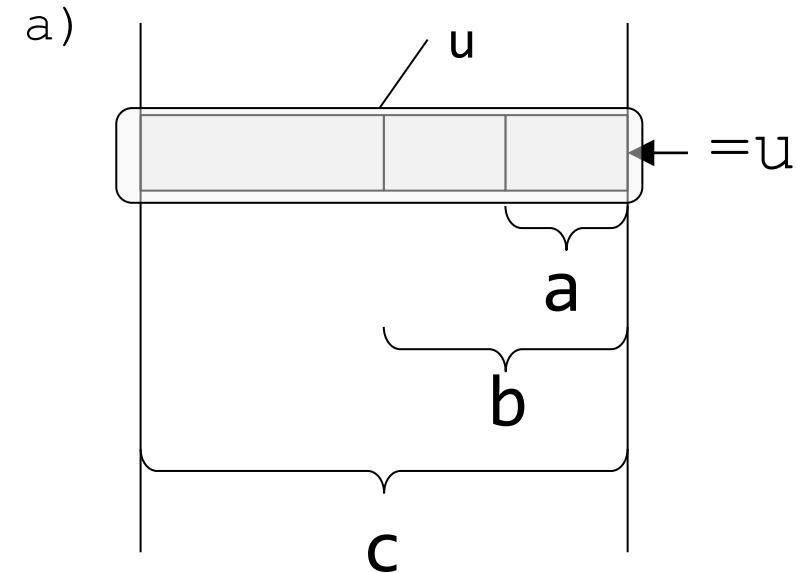
# Union

Storleken (`sizeof(u)`) är garanterad att rymma den största, av de typer, som ingår i unionen.

Vi har deklarationen:

```
union {  
    char  a;  
    short b;  
    long  c;  
} u;
```

```
.ALIGN    2  
offset_a = 0  
offset_b = 0  
offset_c = 0
```



a) Visa hur u representeras i minnet.

Visa kodsekvenser som evaluerar följande uttryck i register R0.

b) `u.a;`

c) `u.b;`

d) `u.c;`

```
b)    LDR    R3, =u  
      LDRB   R0, [R3]  
c)    LDR    R3, =u  
      LDRH   R0, [R3]  
d)    LDR    R3, =u  
      LDR    R0, [R3]
```

# 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);
```

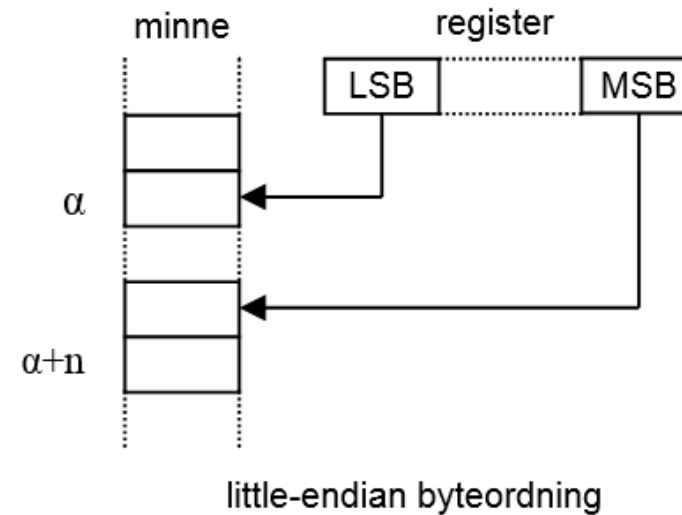
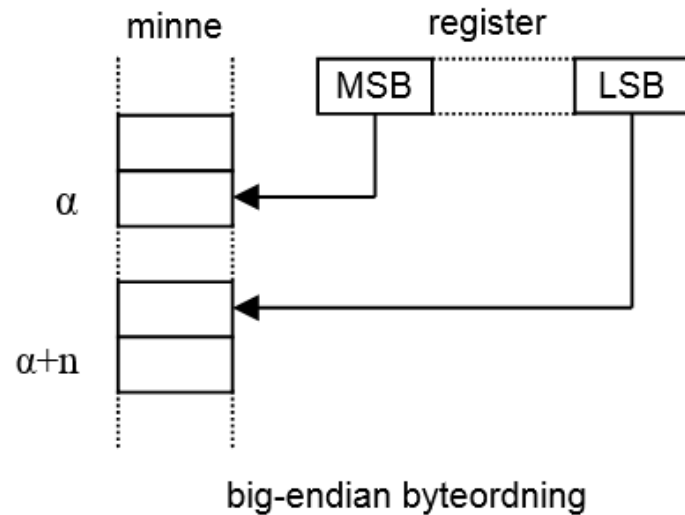
i stället för:

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

| offset | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | mnemonic |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----------|
| 0x10   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | r  | r  | r  | r  | r  | r | r | r | r | r | r | r | r | r | r | GPIO_IDR |

| offset | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  | mnemonic |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----------|
| 0x14   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | GPIO_ODR |

# Byte- och bitordning "endianess"



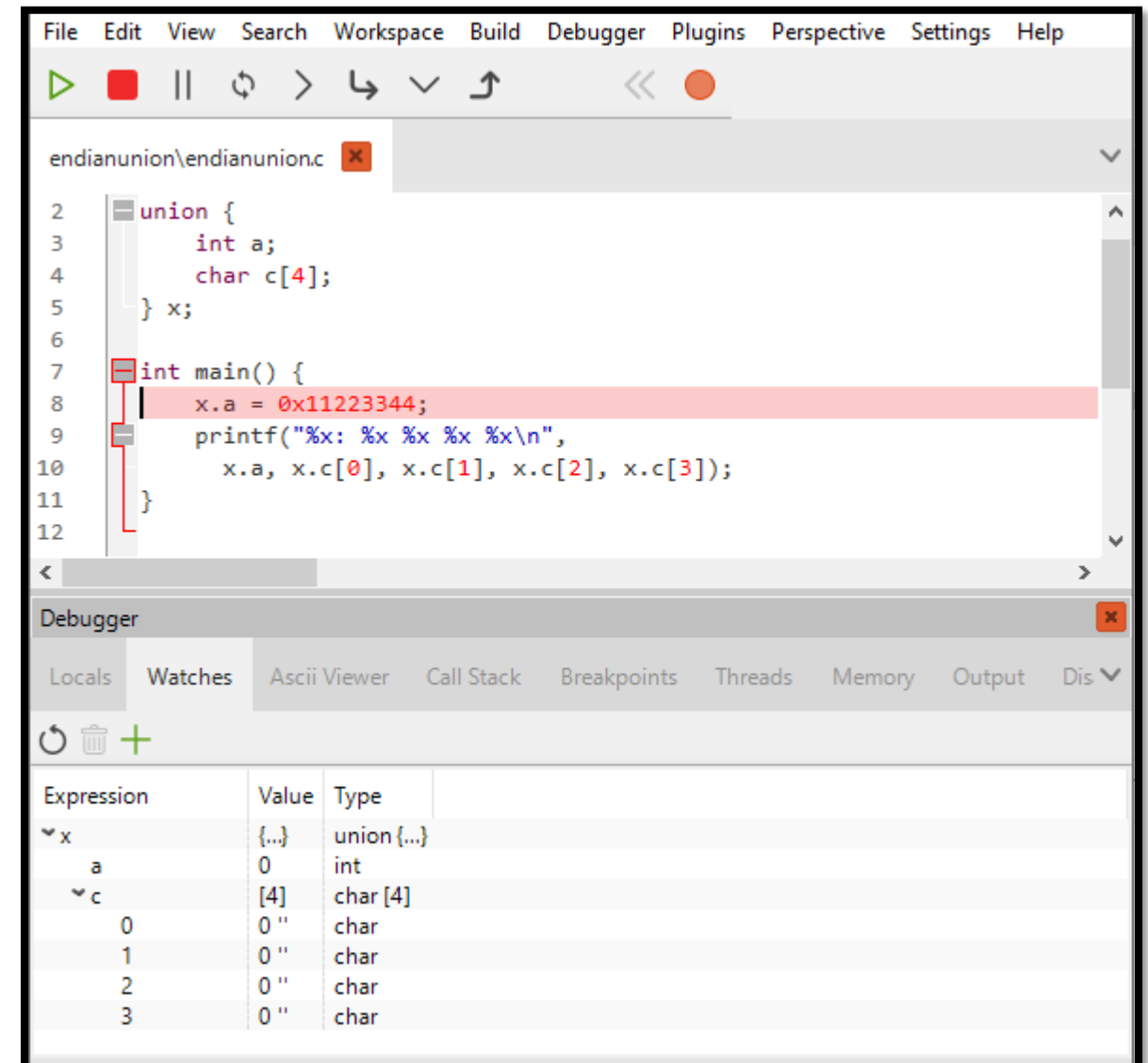
I vårt laborationssystem, ST32F407, använder vi little-endian.

# Byteordning och union

```
#include <stdio.h>

union {
    int a;
    char c[4];
} x;

int main() {
    x.a = 0x11223344;
    printf("%d: %d %d %d %d\n",
        x.a, x.c[0], x.c[1], x.c[2], x.c[3]);
}
```



# Bitfältstyp, ("bitfield")

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

- width är en heltalstyp (`char/short/int/long..`)  $\geq 0$ 
  - 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

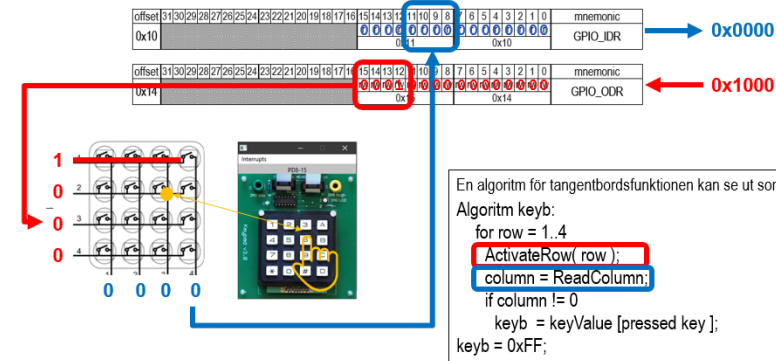
*Exempel:*

```
struct S {  
    unsigned int b1 : 5;  
    unsigned int :0; // börja med ny unsigned  
    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
```

# 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;
    }
}
```

# 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};
```

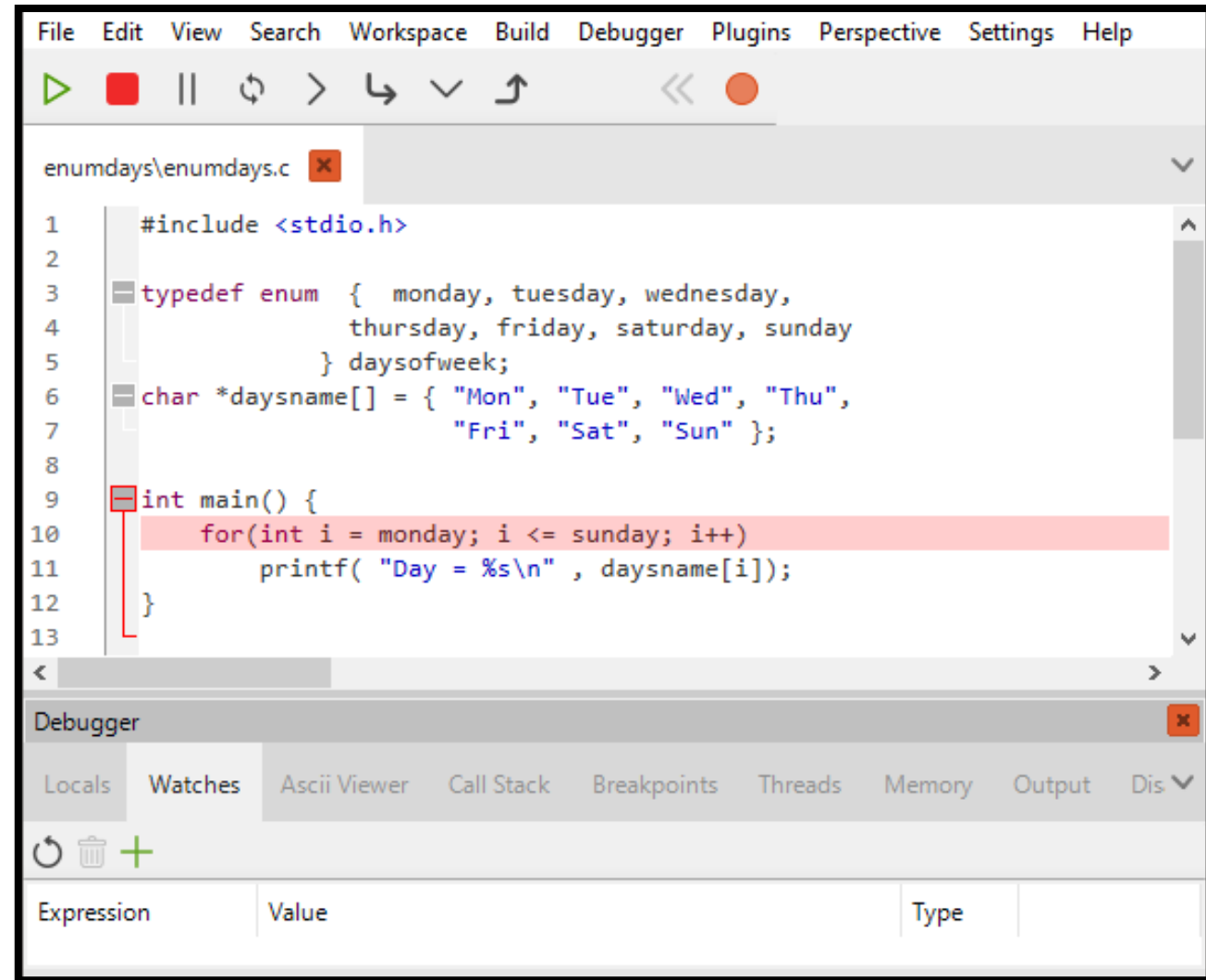
# Uppräkningstyp, enum

```
#include <stdio.h>

typedef enum { monday, tuesday, wednesday,
             thursday, friday, saturday, sunday
} daysofweek;

char *daysname[] = { "Mon", "Tue", "Wed", "Thu",
                     "Fri", "Sat", "Sun" };

int main() {
    for(int i = monday; i <= wednesday; i++)
        printf( "Day = %s\n" , daysname[i]);
}
```



```
enumdays\enumdays.c x
1  #include <stdio.h>
2
3  typedef enum { monday, tuesday, wednesday,
4                thursday, friday, saturday, sunday
5            } daysofweek;
6  char *daysname[] = { "Mon", "Tue", "Wed", "Thu",
7                        "Fri", "Sat", "Sun" };
8
9  int main() {
10     for(int i = monday; i <= sunday; i++)
11         printf( "Day = %s\n" , daysname[i]);
12 }
13
```

Debugger

Locals Watches Ascii Viewer Call Stack Breakpoints Threads Memory Output Dis

| Expression | Value | Type |
|------------|-------|------|
|------------|-------|------|