

Typer och lagringsklasser

Ur innehållet

- Variabler och minnesanvändning

- Mer typer (union/uppräkningsstyp/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

Global synlighet
"global scope"

```
#include <stdio.h>
char x;

int foo()
{
    // x is visible
    // y is not visible
}
char y;
```

Synlig i denna källtext
"file scope"

```
#include <stdio.h>
static char x;

int foo(float x)
{
    /* argument x (float)
       is visible */
}
```

Synlig i denna funktion
"function scope"

```
#include <stdio.h>
static char x;

int foo(float x)
{
    int x;
    /* local x (int)
       is visible */
}
```

Senare deklaration, parameter eller lokal, "döljer" tidigare...

- 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:

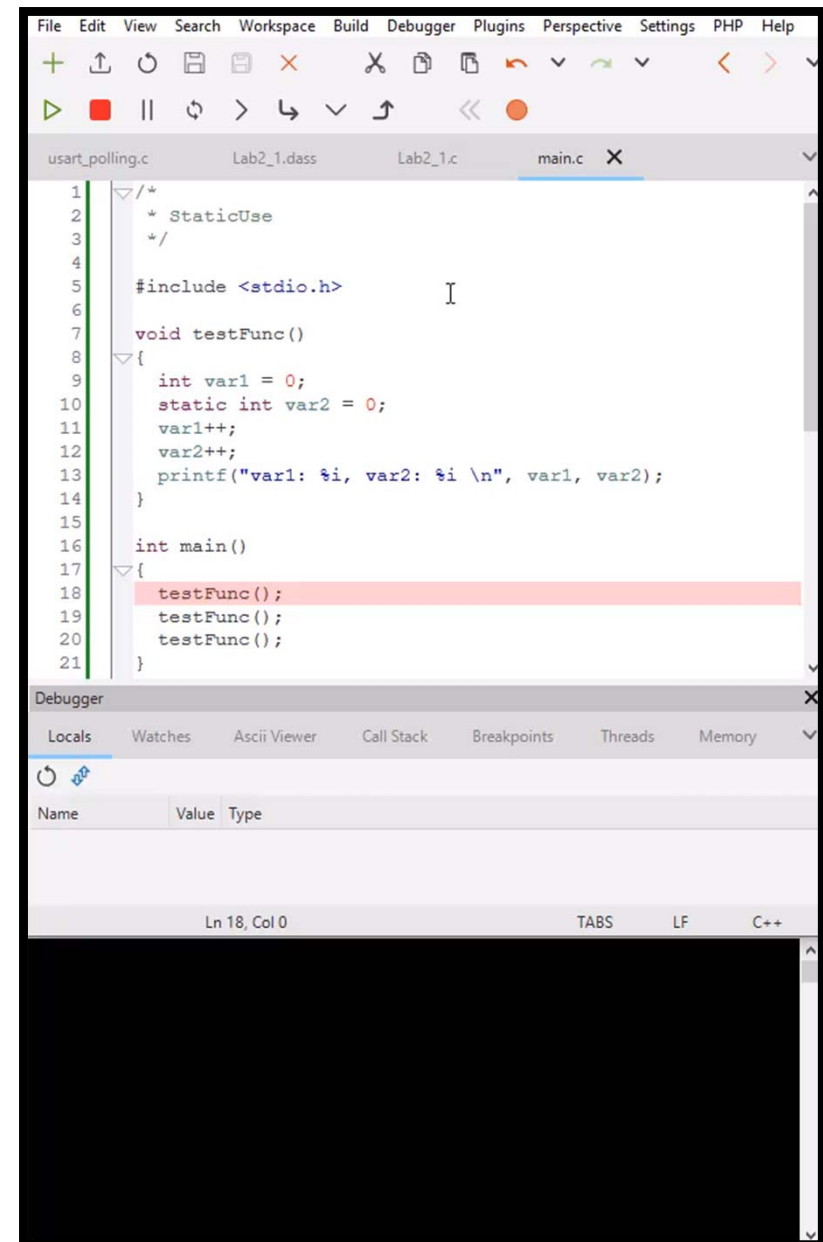
```
@void f (int a, int b, int c )
f:
@{
@    int d:
@    int e;
    PUSH    {R4, R5, LR}
    MOV     R4, R0
    MOV     R5, R1
    ...
    POP     {R4, R5, PC}
@}
```

Permanent/tillfällig lagring

```
#include <stdio.h>

void testFunc()
{
    int var1 = 0;
    static int var2 = 0;
    var1++;
    var2++;
    printf("var1: %i, var2: %i \n", var1, var2);
}

int main()
{
    testFunc();
    testFunc();
    testFunc();
}
```



Synlighet – ("visibility")

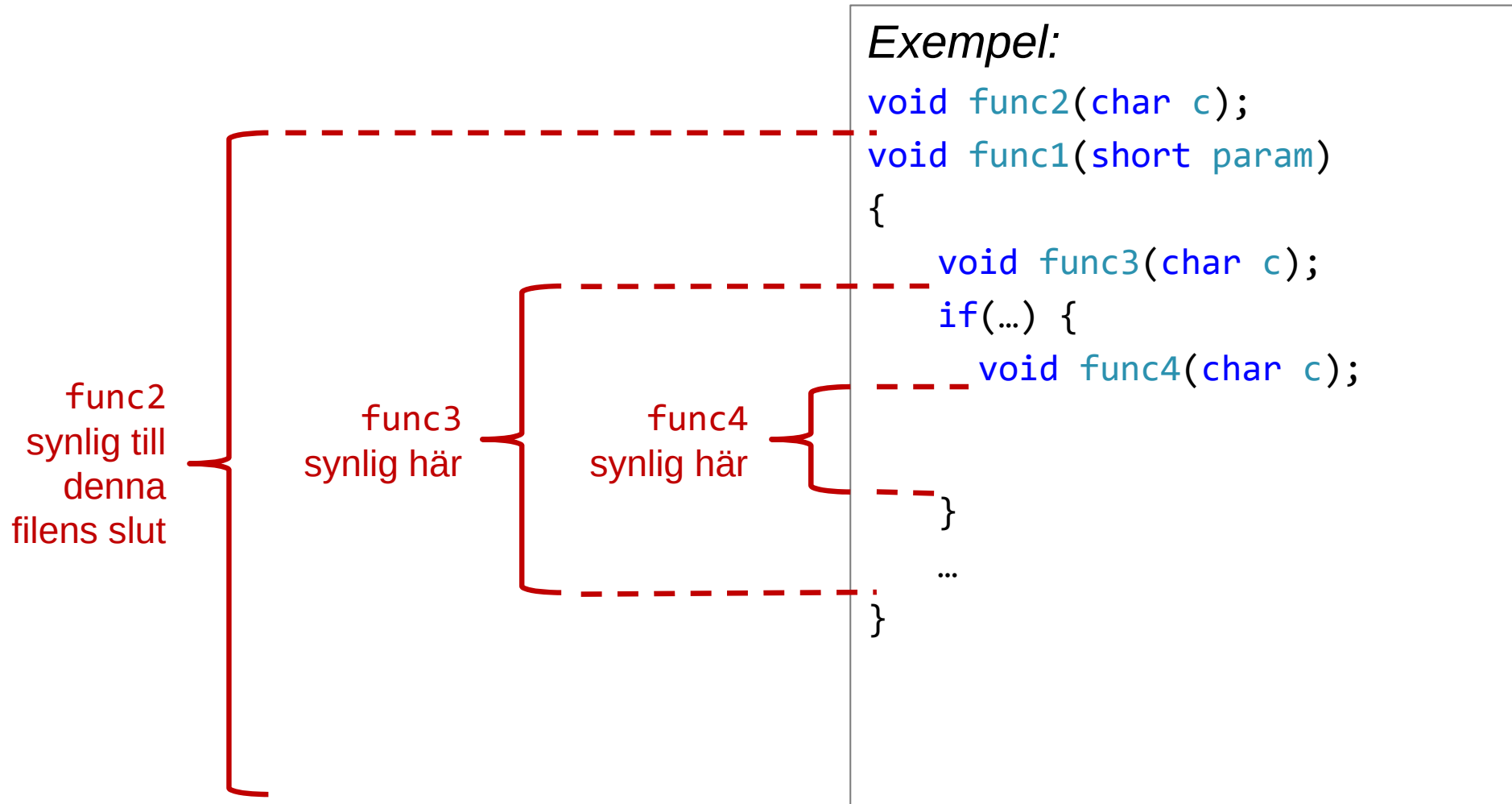
Deklarationer (variabler och funktioner) och uttryck som *typedefs/defines* är synliga först *efter* dess introduktion.

Exempel:

```
void func2(char c);    // Introducera deklaration av func2.
void func2(char c);    // Vi kan upprepa den (flera gånger) ...
void func1(short param)
{
    void func2(char c); // Vi kan också ha den här...
    if(...) {
        void func2(char c); // ... och här ...
        return func2('a');
    }
    ...
    return;
}
```

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.

Synlig i denna funktion
"function scope"

```
#include <stdio.h>

static char x;

int foo(float x)
{
    int x;
    /* local x (int)
       is visible */
}
```

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

Global synlighet
"global scope"

```
#include <stdio.h>

char x;

int foo()
{
    // x is visible
    // y is not visible
}

char y;
```

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

Synlig i denna källtext
"file scope"

```
#include <stdio.h>

static char x;

int foo(float x)
{
    /* argument x (float)
       is visible */
}
```

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

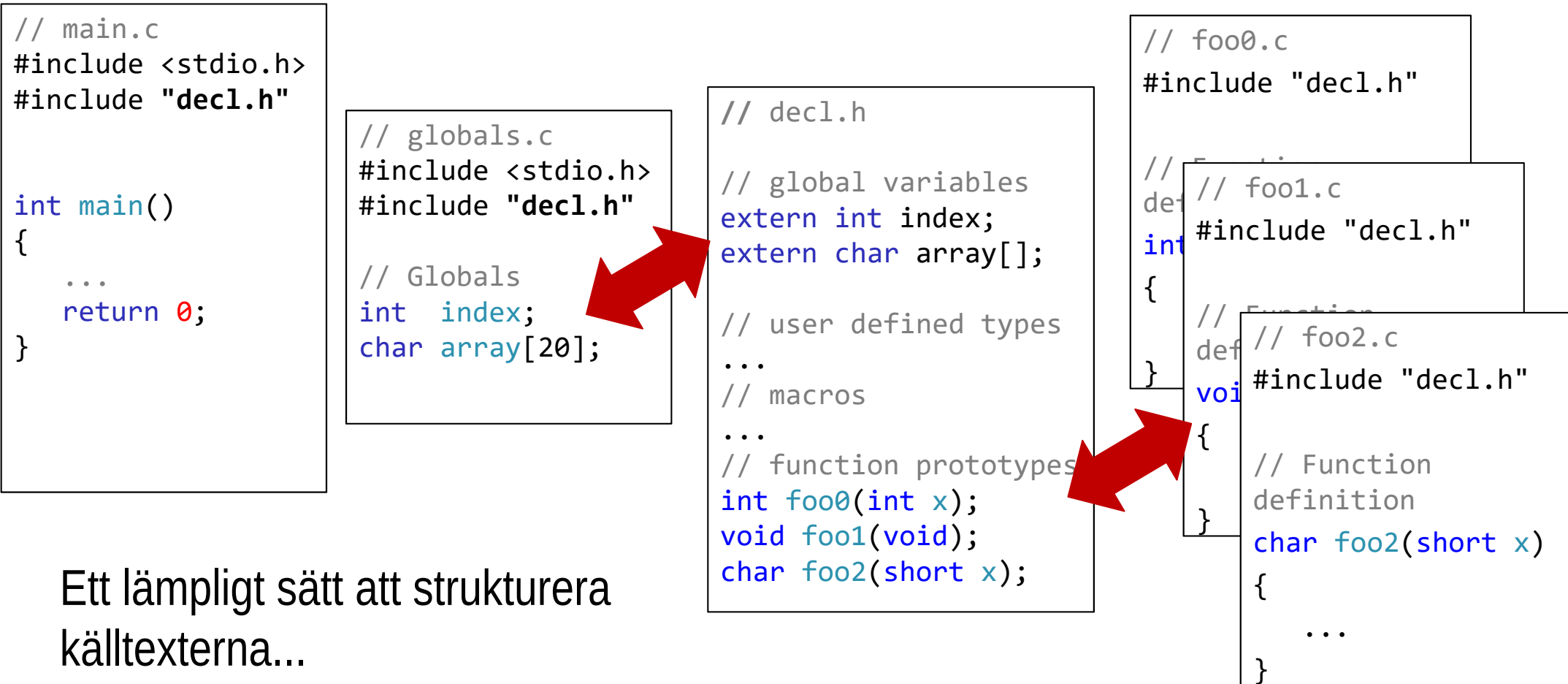
static – permanent lagring intern bindning (dock ej om deklaration i block)

extern – permanent 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;  
}
```

Union

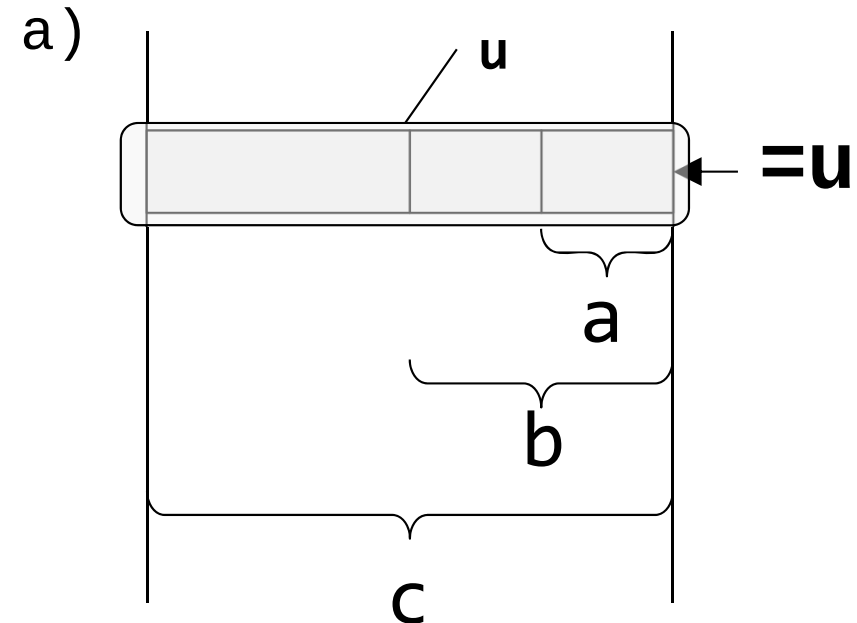
Storleken (`sizeof(u)`) är garanterad att rymma den största, av de typer, som ingår i unionen.

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

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
																		0x11								0x10							

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
																		0x15								0x14							

```
// 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);
```

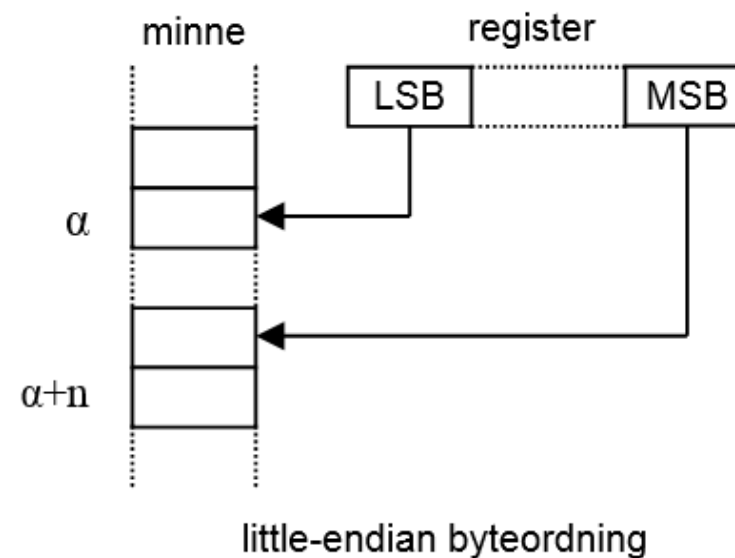
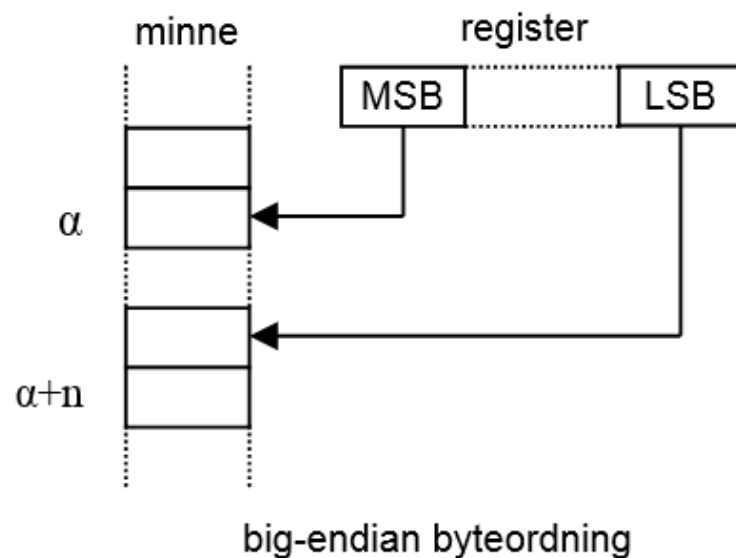
Exempel:

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

i stället för:

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

Byte- och bitordning "endianess"



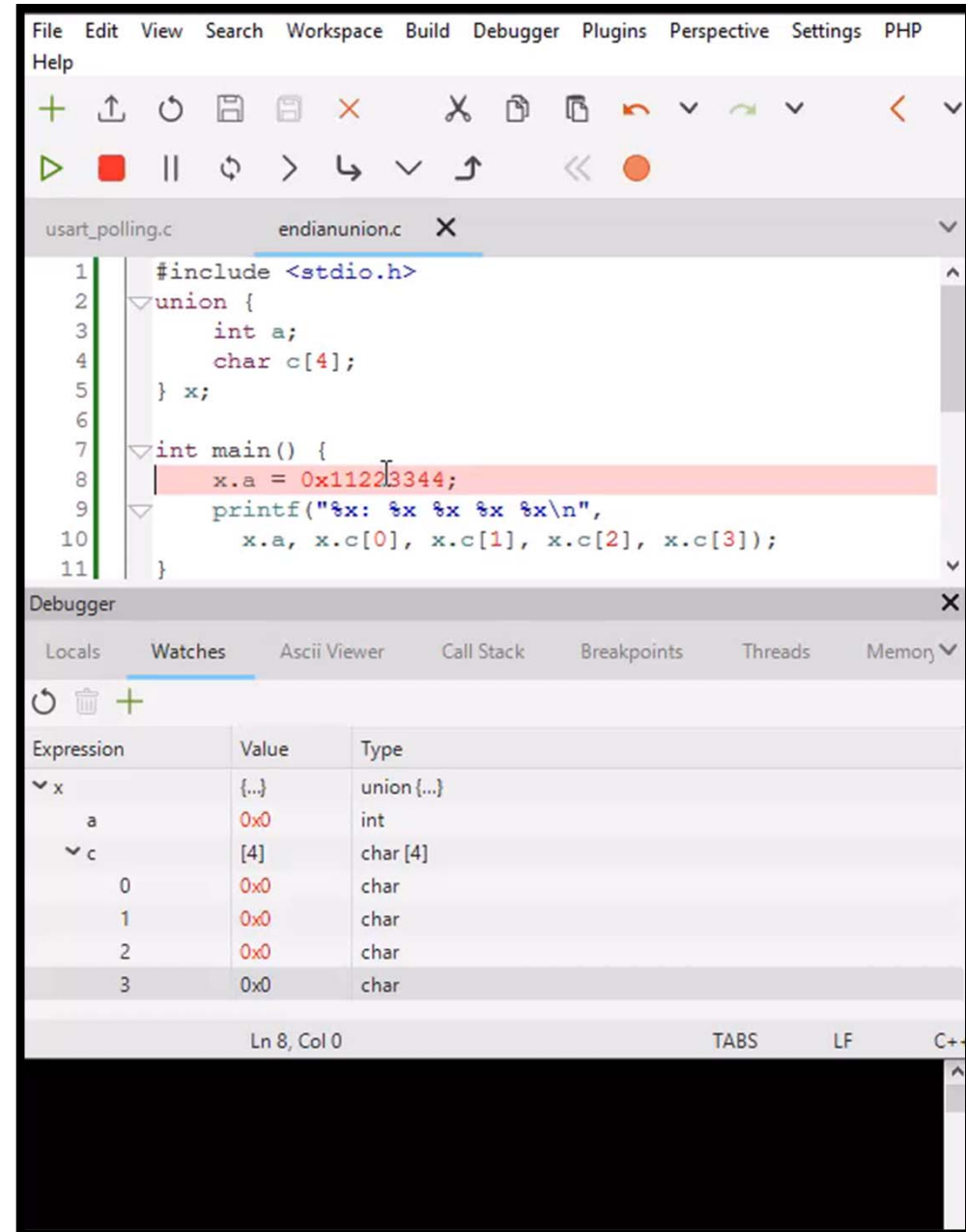
I vårt laborationssystem, ST32F407, använder vi little-endian.

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]);
}
```



The screenshot shows a C code editor with two tabs: `usart_polling.c` and `endianunion.c`. The `endianunion.c` tab is active, showing the following code:

```
1  #include <stdio.h>
2  union {
3      int a;
4      char c[4];
5  } x;
6
7  int main() {
8      x.a = 0x11223344;
9      printf("%x: %x %x %x %x\n",
10         x.a, x.c[0], x.c[1], x.c[2], x.c[3]);
11 }
```

The line `x.a = 0x11223344;` is highlighted in red. Below the code editor is a **Debugger** window with the **Watches** tab selected. It displays the memory layout of the union variable `x`:

Expression	Value	Type
<code>x</code>	<code>{...}</code>	<code>union {...}</code>
<code>a</code>	<code>0x0</code>	<code>int</code>
<code>c</code>	<code>[4]</code>	<code>char [4]</code>
<code>0</code>	<code>0x0</code>	<code>char</code>
<code>1</code>	<code>0x0</code>	<code>char</code>
<code>2</code>	<code>0x0</code>	<code>char</code>
<code>3</code>	<code>0x0</code>	<code>char</code>

The status bar at the bottom indicates the cursor is at `Ln 8, Col 0`, with `TABS` and `LF` line endings, and `C++` language.

Uppräkningsstyp, 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:

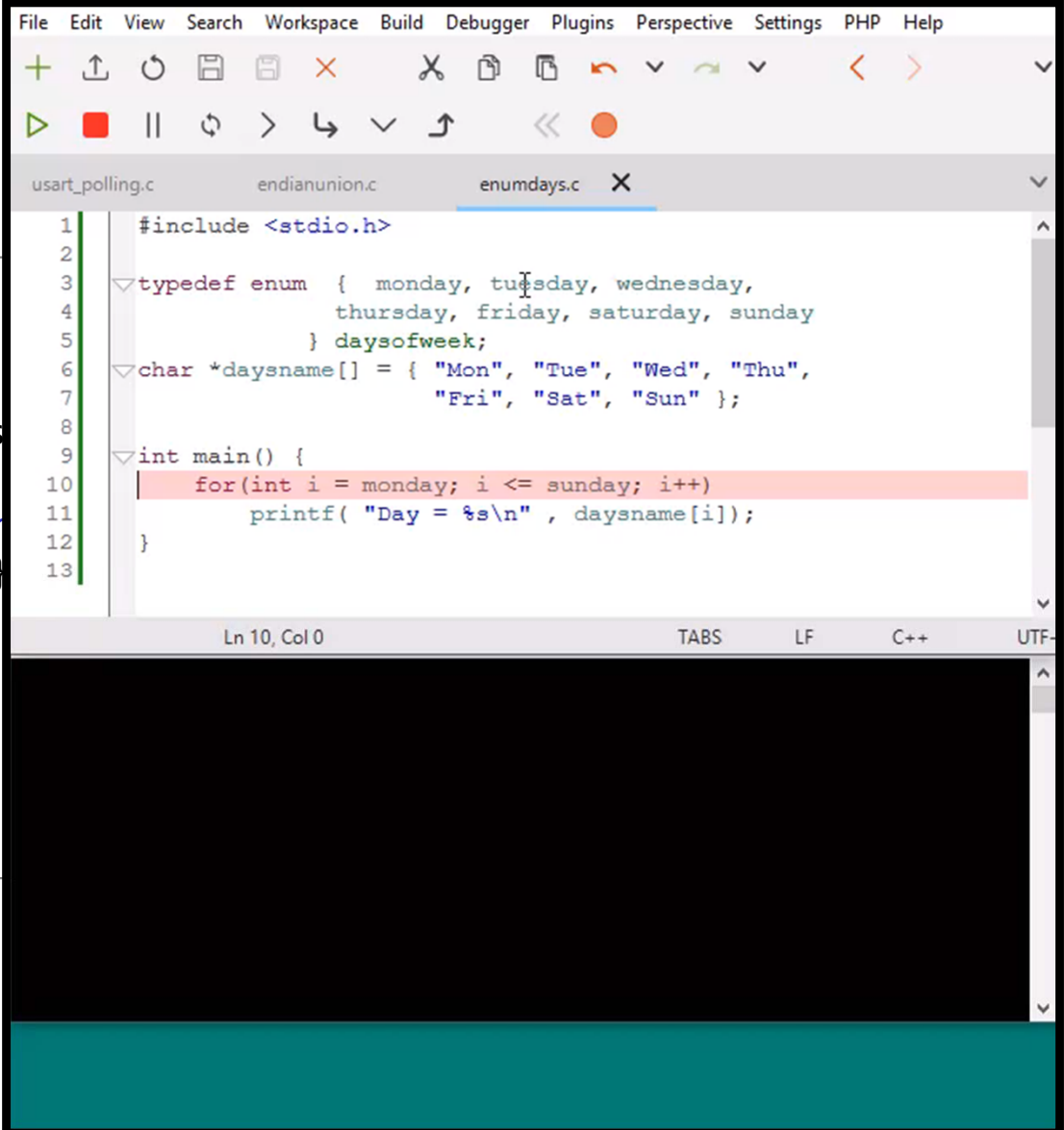
```
enum {B_RS=1, B_RW=2, B_SELECT=4, B_CS1=8, B_CS2=0x10, B_RST=0x20, B_E=0x40};
```

Uppräkningsstyp, 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]);
}
```



The screenshot shows a code editor with the following code in `enumdays.c`:

```
1  #include <stdio.h>
2
3  typedef enum { monday, tuesday, wednesday,
4               thursday, friday, saturday, sunday } daysofweek;
5
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
```

The editor interface includes a menu bar (File, Edit, View, Search, Workspace, Build, Debugger, Plugins, Perspective, Settings, PHP, Help), a toolbar with icons for file operations and execution, and a status bar at the bottom showing "Ln 10, Col 0", "TABS", "LF", "C++", and "UTF-8".

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

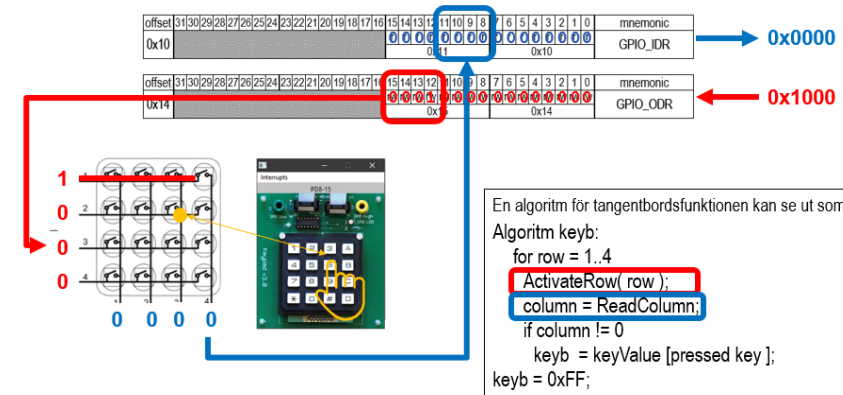
Exempel:

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