



# Kodningskonventioner

Ulf Assarsson

Originalslides av Viktor Kämpe  
(Arbetsboken: sid ~76 – 83.)

# Varför kodningskonventioner?

- Förståelse för
  - Skillnaden mellan lokala/globala variabler.
  - Funktionsargument.
  - Returvärde.
- Möjliggör
  - Mix av assembler och C.



# Funktionsanrop utan argument

```
JSR      delay

delay:
        LDAB      #$FF
loop_delay:
        DECB
        BNE      loop_delay
        RTS
```

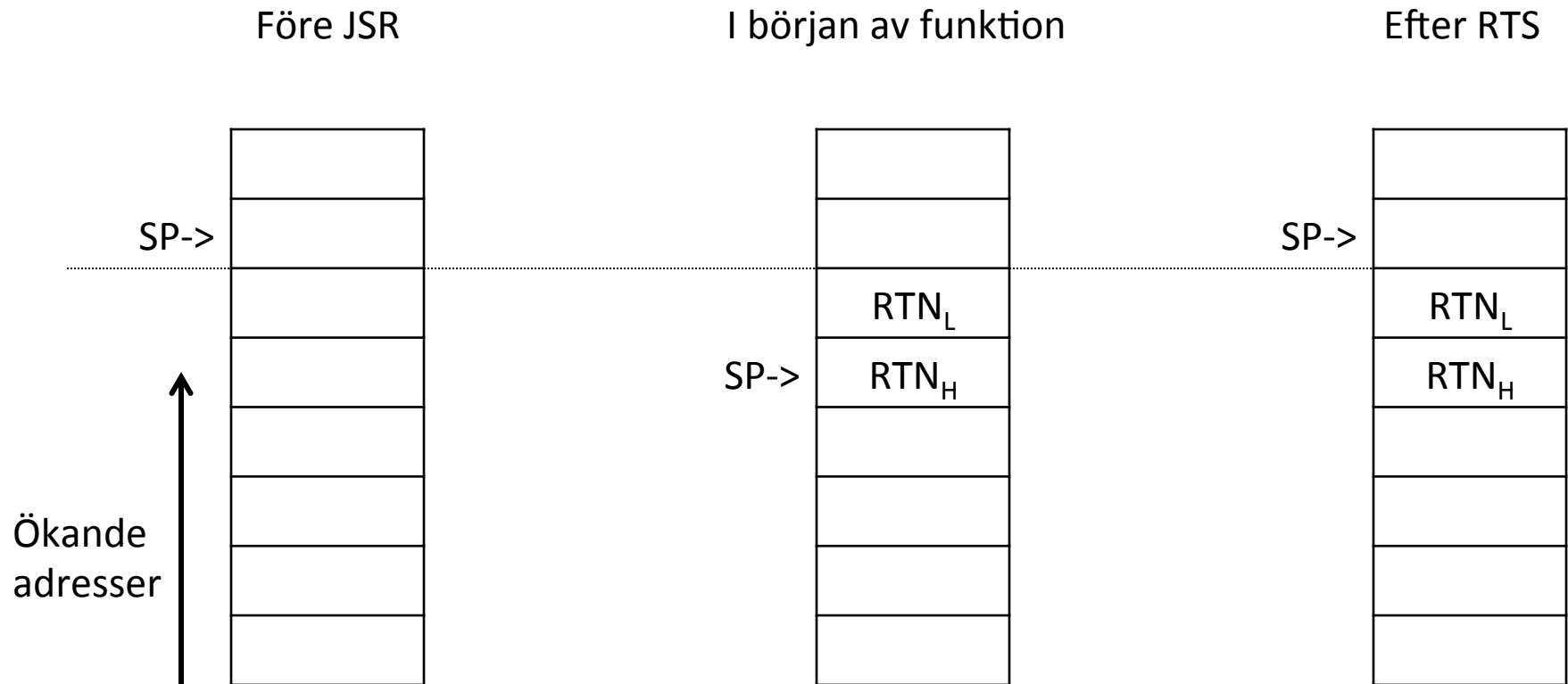
CPU12 Assembler

```
delay();

void delay()
{
    unsigned char tmp = 0xFF;
    do {
        tmp--;
    } while(tmp);
}
```

C – program

# Stackens utseende

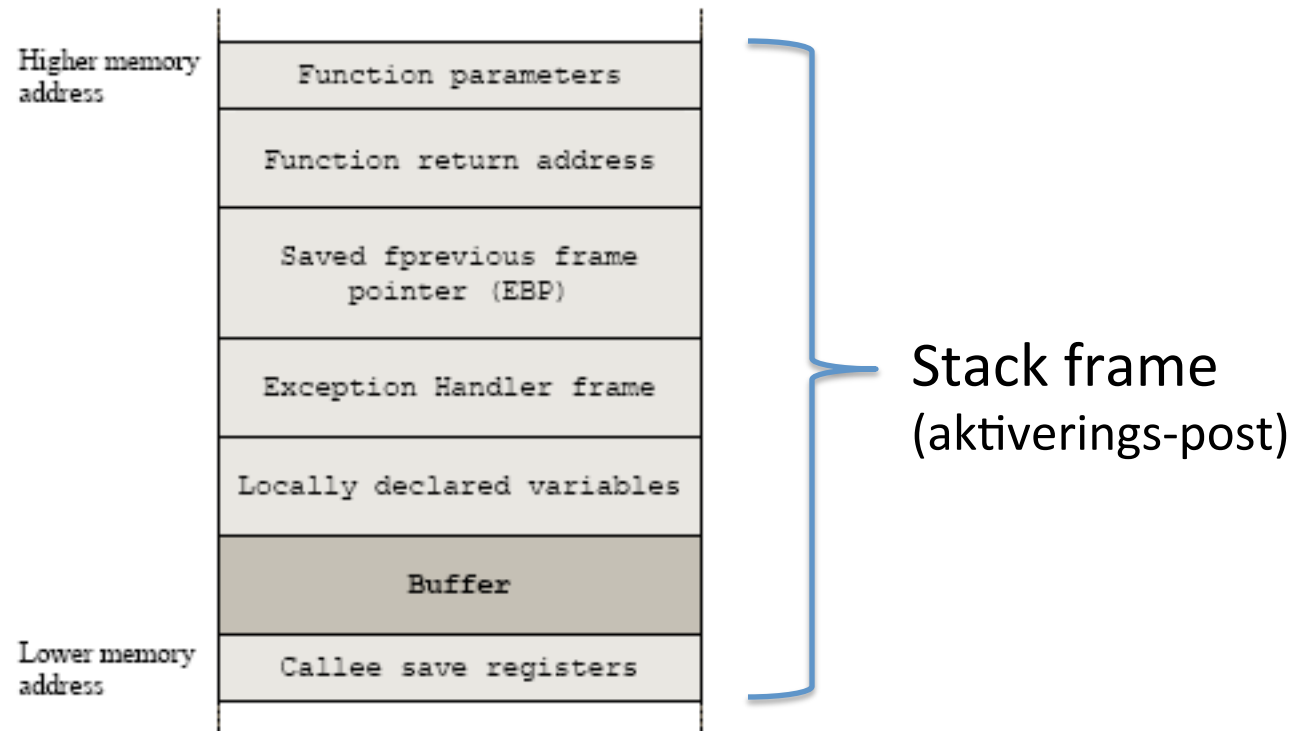


# Funktionsanrop med argument

- Anropskonventioner för XCC12:
  - Kallas också parameteröverföring.
  - Parametrarna överförs via stacken.
    - (Värdet av) argumenten pushas på stacken.
    - Ordningen är höger-till-vänster.

Detaljer i *Arbetsbok för MC12* på sidan 80

# Jfr x86



# Returvärde från funktion

- Returvärdet sker enligt XCC12's konventioner i:
  - Register **D**, om ej beskrivet nedan.
  - Register **B**, om returtyp är `char`.
  - Register **Y/D** om returtyp är `long` eller `float`.
    - MSW i **Y**, LSW i **D**.
  - Via (en pekare till) minnet om returtyp är en `struct`.

*Se Arbetsbok för MC12 på sidan 81.*

# Ett enkelt C-program

```
#include <stdio.h>

int g_var;    //global synlighet

int myAdd(int x, int y)
{
    // lokal variabel
    int sum;
    sum = x+y;
    return sum;
}

int main()
{
    // lokala variabler
    int var1, var2;

    var1 = 4;
    g_var = 1;

    var2 = myAdd(g_var, var1);

    return 0;
}
```

Funktionsdeklaration

I följande slides använder vi konventioner för XCC12 och 16 bitars `int`

Funktionsanrop



# Passa första parametern

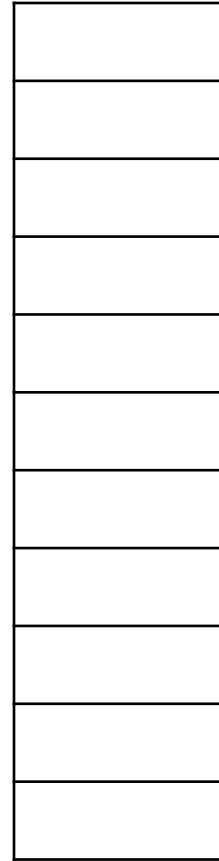
```
int myAdd(int x, int y)
{
    // lokal variabel
    int sum;
    sum = x+y;
    return sum;
}
```

```
var2 = myAdd(g_var, var1);
```

Push

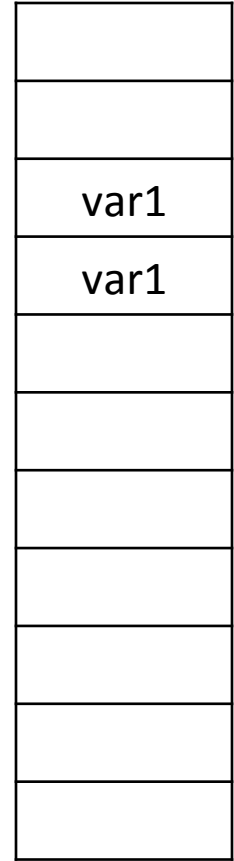
Före

SP->



Efter

SP->



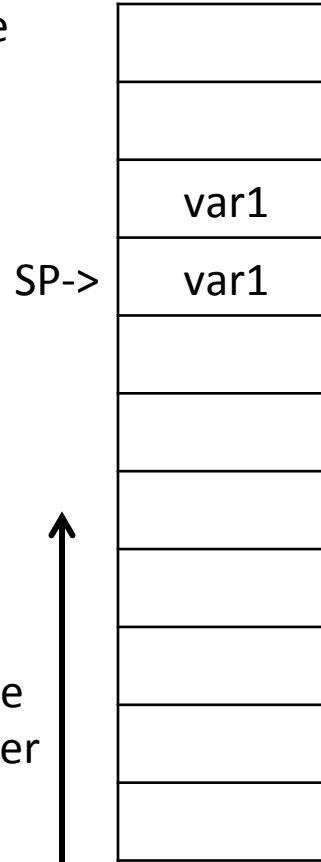
# Passa andra parametern

```
int myAdd(int x, int y)
{
    // lokal variabel
    int sum;
    sum = x+y;
    return sum;
}
```

```
var2 = myAdd(g_var, var1);
```

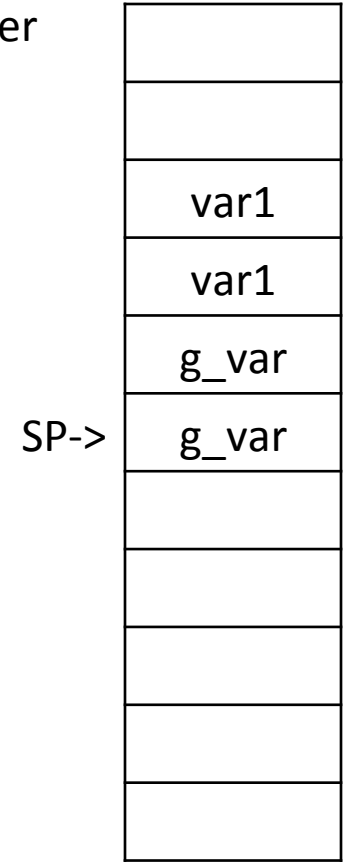
Push

Före



Ökande adresser

Efter



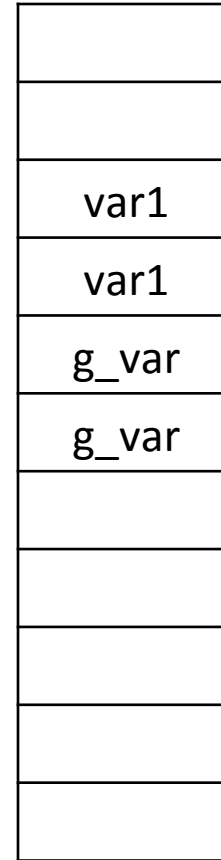
# Spara returadress på stacken

```
int myAdd(int x, int y)
{
    // lokal variabel
    int sum;
    sum = x+y;
    return sum;
}
```

```
var2 = myAdd(g_var, var1);
```

JSR

Före



Efter



# Lokala variabler på stacken

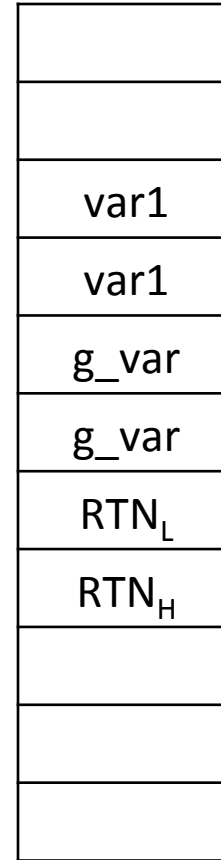
- Lokala variabler ligger på stacken.
- Utrymme skapas genom att flytta stack pekaren (SP).

# Lokala variabler på stacken

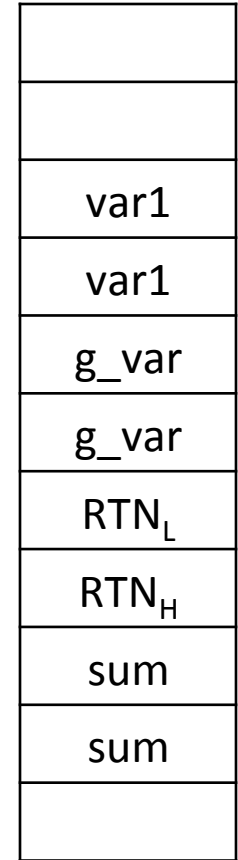
```
int myAdd(int x, int y)
{
    // lokal variabel
    int sum;
    sum = x+y;
    return sum;
}

var2 = myAdd(g_var, var1);
```

Före



Efter



SP->

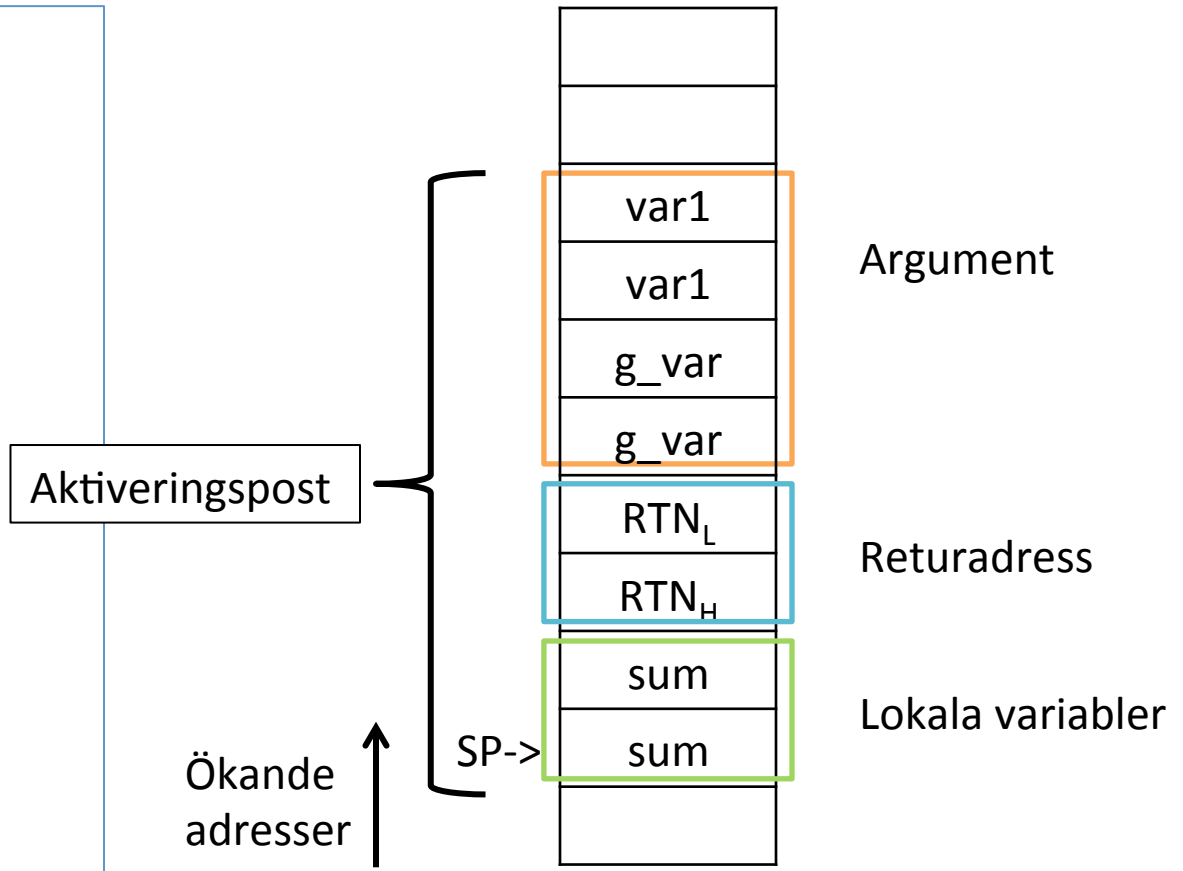
SP->

Ökande  
adresser ↑

# Aktiveringspost / Stack frame

```
int myAdd(int x, int y)
{
    // lokal variabel
    int sum;
    sum = x+y;
    return sum;
}
```

```
var2 = myAdd(g_var, var1);
```



# Prolog

- Prologen av en funktion skapar utrymmet för lokala variabler
- På CPU12 så görs det genom att flytta SP
- LEAS
  - Load Effective Address to SP
  - T ex: LEAS -4,SP // om **två** 16-bits lokala variabler

# Epilog

- Epilogen av en funktion återlämnar minnet för lokala variabler.
  - Görs på motsvarande sätt som prologen.
    - Ex:
      - LEAS 4,SP
- RTS





[myAdd till assembler för hand]

# Recept för ett funktionsanrop

1. Pusha argumenten på stacken.
2. JSR (pusha återhoppadressen på stacken).
3. Prolog: skapar utrymme för lokala variabler.
4. { Funktionskroppen }
5. Lägg returvärde i rätt register.
6. Epilog: återlämnar utrymme för lokala variabler.
7. RTS (pop av programräknaren PC)
8. Återställ stacken till tillståndet innan argumenten pushades.



[assemblerkod från kompilator]

# Kombinera C och assembler

- Anropa assembler-rutiner från C
- Anropa C-rutiner från assembler
- Kodgenerering sker från olika filer, så "ihopkopplingen" sker i länkningsen.
  - ".o12" filer

# XCC genererar assembler från C

- Från C till assembler
- Från assembler till maskinkod
- Symbolerna i assemblern är samma som i C fast med ett understreck:

`myAdd()`            i C

`_myAdd:`            i assembler

# Import

**IMPORT** i assembler säger att symbolen kommer från en annan fil.

```
IMPORT
```

```
_main
```



Symbol från annan fil ( t ex .c-fil)

```
...
```

```
JSR
```

```
_main
```



Börja exekvera main-funktionen

# Extern

**extern** i C säger att symbolen kommer från en annan fil.

```
#include <stdio.h>
```

```
extern int g_var;
```


```
void myCLI();
```


```
void main()
```

```
{
```

```
    myCLI();
```

```
}
```

 g\_var definerad i annan fil ( t ex .c-fil)

 Extern ej nödvändig för funktioner, prototypen säger: finns vid länkning.

```
EXPORT    _myCLI
_myCLI:
    CLI
    RTS
```

# Synlighet av symboler

- Alla symboler synliga per-default i C
- Inga symboler synliga per-default i assembler
  - Gör symboler (i .s12 filen) synliga med **EXPORT**
  - Gör symboler från andra filer synliga med **IMPORT**



# static

`static` i C kan göra två saker

1. Ta bort synlighet (från andra filer) av symboler.
2. Allokera minne för lokala variabler som om de vore globala variabler (men fortfarande med lokal synlighet).

# Static variant 1

```
static int var;
```

Variabeln *var* kan ej ses från andra filer, oavsett om man använder `extern` eller `IMPORT`.

# Static variant 2

```
#include <stdio.h>

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

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

Utskrift:

```
var1: 1, var2: 1
var1: 1, var2: 2
var1: 1, var2: 3
```

var2 initialiseras till noll endast första gången vi anropar funktionen, men behåller sedan sitt värde mellan anrop.

# Static variant 2 – vanligt exempel

```
#include <stdio.h>

void testFkt()
{
    static int bFirstTime = 1;
    if(bFirstTime) {
        // T ex allokeras minne på heapen
        // eller förberäkna något.
        bFirstTime = 0;
    }
    ... Gör huvudberäkningarna.
}

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

**OBS – koden är inte trådsäker!**

T ex: rita gubbe, skriv ut resultat i ett fönster, gör avancerad AI-simulering, skicka data över nätverk.

# Några fler qualifiers

- inline
- const
- static
- extern
- volatile
- (restrict)

# Include guards

```
// vecmath.h
#ifndef VECMATH_H
#define VECMATH_H
```

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

```
typedef struct {
    Vec2f min;
    Vec2f max;
} Box2f; // box
```

```
static inline Box2f Add(Box2f * box, Vec2f v) // Adds vector v to min and max of box
{
    Box2f res;
    res.min.x = box->min.x + v.x;
    res.min.y = box->min.y + v.y;
    res.max.x = box->max.x + v.x;
    res.max.y = box->max.y + v.y;
    return res;
}
```

```
#endif //VECMATH_H
```

Funktionen Add skulle bli definierat för varje fil som inkluderar den, om det inte stod "static". Snyggare är oftast dock att lägga funktionsdefinitionern i en .c-fil (vecmath.c)

# Typer

**#define** – find and replace på textnivå. Typedef funkar inte på samma sätt (men ofta ingen praktisk skillnad)

- **typedef** int postnr;  
**typedef** int gatunummer;
  - postnr x = 41501;
  - Gatunummer y = 3; // Observera att x och y nu har samma typ så man kan skriva x = y;

**Angående typer:** Läs från **höger till vänster:** (const finns sedan C89/90)

- unsigned char \* const inport = (unsigned char\*) 0x400; // inport är en constant pekare till en unsigned char
- unsigned char const \* inport = (unsigned char\*) 0x400; // inport är en pekare till en constant unsigned char
- const unsigned char \* inport = (unsigned char\*) 0x400; // samma som ovan, dvs inport är en pekare till en const  
// unsigned char
- char unsigned const \* inport3 = (unsigned char\*) 0x400; // inport är en pekare till en constant unsigned char
- char unsigned const \* const inport4 = (unsigned char\*) 0x400; // inport är en constant pekare till en constant unsigned  
// char

MEN...

- typedef unsigned char\* port8ptr;
- const port8ptr p = (port8ptr)0x400;
- const unsigned char\* p2 = (port8ptr)0x400;
- p2=0; // OK för p2 är pekare till const unsigned char
- p = 0; // ej OK för p är const-pekare till unsigned char
- “port8ptr const p3” är samma som const port8ptr p3”

# Type “Union”

```
union {  
    int d;  
    char f;  
};  
d = 4;  
f = 'i';
```

```
typedef struct {  
    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;
```



# Viktiga koncept

- Aktiveringspost
- Prolog/Epilog av funktion
- Keywords:
  - Static (C)
  - Extern (C)
  - Import, Export (assembler)

# Er hemuppgift. Lösning

```
int main()
{
    char s1[] = "Emilia";
    char **pp, *p;
    p = &s1[0];           // p = adressen till 'E'

    // eller
    p = s1;              // dvs p pekar på s1. Eller, p pekar på 'E'.

    pp = &p;            // pp pekar på p
    **pp = 'A'         // dubbel avreferering av "pp som pekar på p som pekar på 'E'".
    // eller
    (*pp)[0] = 'A';    // *pp pekar på p. Så det blir: "p[0]" = 'E'

    *(*pp + 2) = 'e'; // *(p + 2). Dvs innehållet i (p + 2) tilldelas 'e'.

    printf("s1 = %s", s1);
    getch();
    return 0;
}
```

# Uppgifter:

- Definiera en struct. Accessa en structmedlem via en pekare till structen. Dvs “->”-notation.
- Skapa en funktionspekare till en funktion som tar en int-parameter och returnerar en float.
  - Anropa funktionen och tilldela värdet till variabel a.
- I assembler, gör ett anrop till en funktion som tar 2 ints och returnerar summan av dem, i reg Y/D.
  - msw i Y. lsw i D.

# Uppgifter:

- Definiera en struct. Accessa en structmedlem via en pekare till structen. Dvs “->”-notation.
- Skapa en funktionspekare till en funktion som tar en int-parameter och returnerar en float.
  - Anropa funktionen och tilldela värdet till variabel a.
- I assembler, gör ett anrop till en funktion som tar 2 ints och returnerar summan av dem, i reg Y/D.
  - msw i Y. lsw i D.

# Uppgifter:

## Recept för ett funktionsanrop

1. Pusha argumenten på stacken.
  2. JSR (pusha återhopsadressen på stacken).
  3. Prolog: skapar utrymme för lokala variabler.
  4. { Funktionskroppen }
  5. Lägg returvärde i rätt register.
  6. Epilog: återlämnar utrymme för lokala variabler.
  7. RTS (pop av programräknaren PC)
  8. Återställ stacken till tillståndet innan argumenten pushades.
- I assembler, gör ett anrop till en funktion som tar 2 ints och returnerar summan av dem, i reg Y/D.
    - msw i Y. lsw i D.