Grundläggande C-programmering – del 3
Structs och funktionspekare

Ulf Assarsson

Läromoment:

- Structs, pekare till structs (pilnotation), array av structs,
- Portadressering med structs
- Funktionspekare,
- structs med funktionspekare (objektorienterad stil)

Kopplat till Arbetsbok

- Avsnitt 4: GPIO
- Avsnitt 5: ASCII-display

Hemuppgifter: v3.

Föregående lektion

- Pekare

  int a;

  int*p = &a;

- Absolutadressering (portar):
  - typedef, volatile, #define

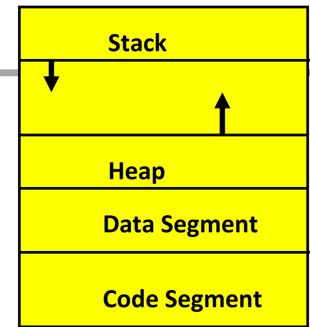
```
typedef volatile unsigned char* port8ptr;
#define INPORT_ADDR 0x40011000
#define INPORT *((port8ptr)INPORT_ADDR)

// läser från 0x40011000
value = INPORT;
```

- Arrayer av pekare, arrayer av arrayer

```
char *fleraNamn[] = {"Emil", "Emilia", "Droopy"};

int arrayOfArrays[3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
```



Vanlig fråga...

Ska man skapa en pekare eller en array (när det gäller specialfallet sträng)?

```
1.  char* s1 = "Emilia";  // skrivskyddat strängliteralminne (datasegmentet)
2.  char s2[] = "Emilia"; // skrivbart minne (datasegmentet eller stacken)
```

Spelar ofta ingen roll, men ska du ändra strängen så välj nr 2.

// skrivskyddat strängliteralminne:

```
void main()
{
    char *a;
    a = "hejsan”;
}
```

```
const static char literal_constant_34562[7] = {’h',
'e', ’j', ’s', ’a', ’n', '\0'};

int main()
{
    char *a;
    a = &literal_constant_34562[0];

}
```

Structs

Sammansatta datatyper (structs)

En så kallad `struct` (från eng. *structure*)

- (Kallas även ”post” på svenska)
- Har en/flera medlemmar.
- Medlemmarna kan vara av t ex:
  - bas-typ
    - `char, short, int, long, long long` (som signed/unsigned)
    - `float, double`
  - Sammansatta typer (t ex en annan `struct`).
  - Pekare (även till funktioner och samma `struct`)
  - Arrayer
  - Kombinationer av ovanstående

Användning av struct

```
#include <stdio.h>

char* kursnamn = ”Maskinorienterad Programmering";

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

int main()
{
    struct Course mop;

    mop.name = kursnamn;
    mop.credits = 7.5f;
    mop.numberOfParticipants = 110;

    return 0;
}
```

Definition av strukturen

Deklaration av variabeln mop

Access till medlemmar via .-operatorn

Initieringslista

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

struct Course kurs1 = {"MOP", 7.5f, 110};
struct Course kurs2 = {"MOP", 7.5f};
```

← initieringslista

En `struct` **kan** initieras med en initieringslista. Initieringen sker i samma ordning som deklarationerna, men alla medlemmar måste inte initieras.

Typedef – alias för typer

Exempel:

```
typedef unsigned int uint32, uint32_t;

typedef signed short int int16;

typedef unsigned char *ucharptr;

uint32 a, b = 0, c;
int16 d;
ucharptr p;
```

```
typedef int postnr;
typedef int gatunr;
postnr x = 41501;
gatunr y = 3;
x = y; // Ingen typkonvertering behövs ty bådas
typer är int.

// Lurighet: '*' ingår inte i typdeklarationen för byteptr2
typedef char* byteptr, byteptr2; // byteptr2 fel!
typedef char *byteptr, *byteptr2; // rätt
typedef char *byteptr, byte; // rätt
```

`typedef` förenklar/förkortar uttryck, vilket kan öka läsbarheten.

```
typedef unsigned char  uint8, …;
```

typ alias/typnamn

CHALMERS

Structs (sammansatta datatyper)

Sv: Poster

```
        optional
struct StructName {
    typ medlem1;
    typ medlem2;
    …
} variabel1, variabel2 … ;
          optional
```

```
Ekvivalenta skrivsätt:

struct Player { // “Player” kan här skippas
    int age;
    char* name;
} player1, player2;

Eller:

struct Player {
    int age;
    char* name;
};
struct Player player1, player2;

Eller:

typedef struct tPlayer {// tPlayer kan skippas
    int age;
    char* name;
} Player ;
Player player1, player2;
```

Structs (sammansatta datatyper)

Initieringar av structs

Vanligt förekommande:

```
typedef struct {
   int age;
   char* name;
} Player;

//Player är nu typalias för
denna struct. Fördel: slipper
skriva ”struct Player”

Player player1 = {15, "Ulf"};
Player player2 = {20, "John Doe"};
// eller t ex
player1.age = 16;
player1.name = "Striker";
```

Structs kan innehålla andra structs:

```
typedef struct {
   int x;
   int y;
} Position;

typedef struct {
   int      age;
   char*    name;
   Position pos;
} Player;

Player player1 = {15, "Striker”, {5, 10}};
// eller t ex
player1.pos.x = 6;
player1.pos.y = 11;
player1.pos = (Position){6,11};// funkar

// Ofullständing initiering OK!
Player player1 = {15, "Striker”, {5}};
```

Structs (sammansatta datatyper)

I kap 5, s 112:

```
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;
```

Skapa och initiera en variabel av typen GEOMETRY:

```
GEOMETRY ball_geometry = {
    12, 4, 4,
    { //  POINT px[20]
          {0,1}, // px[...]
          {0,2},
          {1,0},
          {1,1},
          {1,2},
          {1,3},
          {2,0},
          {2,1},
          {2,2},
          {2,3},
          {3,1},
          {3,2} // Här utnyttjar vi ofullständig
    }           // initiering (12 av 20)
};
```

Pekare till struct - pilnotation

```
    Course *pmop; // Pekare till struct

    (*pmop).name = …
```

Eller enklare:

```
    pmop->name   = …
```

Pilnotationen förenklar, då det är vanligt att ha pekare till structs

Pekare till struct

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

char* kursnamn = ”Maskinorienterad Programmering";

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

int main()
{
     Course *pmop; // Pekare till struct
     pmop = (Course*)malloc(sizeof(Course));

     (*pmop).name = kursnamn;
     pmop->name   = kursnamn;
     pmop->credits = 7.5f;
     pmop->numberOfParticipants = 110;

     free(pmop);
     return 0;
}
```

I Java:

```
public class Course {
   String name;
   float credits;
   int numberOfParticipants;
}

Course mop = new Course();
mop.name = …
mop.credits = 7.5;
…
```

```
// eller
Course mop, *pmop;
pmop = &mop;
// och givetvis utan free()
```

Access till medlemmar via -> operatorn

Java har lurat er…

```
Dog myDog = new Dog("Rover");
```

myDog är (under ytan) en pekare till en Dog.

Pekare behövs oftast i ett program. Men ett programspråk kan dölja behovet genom språkets konstruktion.

Array av structs

```
#include <stdio.h>

typedef struct {
	char* name;
	float credits;
} Student;

Student stud1 = {"Per", 200.0f};
Student stud2 = {"Tor", 200.0f};
Student stud3 = {"Ulf",  20.0f};

Student students[3] = {stud1, stud2, stud3};
// eller bara:
Student students[] = {{"Per", 200.0f}, {"Tor", 200.0f}, {"Ulf",  20.0f}};

int main()
{
	printf("%s, %s, %s\n", students[0].name, students[1].name, students[2].name);

	return 0;
}
```

```
Per, Tor, Ulf
```

Portadressering med structs

Istället för:

```
#define     portModer   ((volatile unsigned int *)  (PORT_DISPLAY_BASE))
#define     portOtyper  ((volatile unsigned short *)(PORT_DISPLAY_BASE+0x4))
#define     portOspeedr ((volatile unsigned int *)  (PORT_DISPLAY_BASE+0x8))
#define     portPupdr   ((volatile unsigned int *)  (PORT_DISPLAY_BASE+0xC))
#define     portIdrLow  ((volatile unsigned char *) (PORT_DISPLAY_BASE+0x10))
#define     portIdrHigh ((volatile unsigned char *) (PORT_DISPLAY_BASE+0x11))
#define     portOdrLow  ((volatile unsigned char *) (PORT_DISPLAY_BASE+0x14))
#define     portOdrHigh ((volatile unsigned char *) (PORT_DISPLAY_BASE+0x14+1))
```

Så kan vi utnyttja structs genom att skriva:

```
typedef struct {
    uint32_t    moder;
    uint16_t    otyper;      // +0x4
    uint16_t    otReserved;
    uint32_t    ospeedr;     // +0x8
    uint32_t    pupdr;       // +0xc
    uint8_t     idrLow;      // +0x10
    uint8_t     idrHigh;     // +0x11
    uint16_t    idrReserved;
    uint8_t     odrLow;      // +0x14
    uint8_t     odrHigh;     // +0x15
    uint16_t    odrReserved;
} GPIO;
```

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

Exempel:
GPIO_E.moder    =  0x55555555;
GPIO_E.otyper   =  0x00000000;
GPIO_E.ospeedr  =  0x55555555;
GPIO_E.pupdr    &= 0x55550000;
```

Portadressering – individuella bytes

Nyss definierade vi idrLow och idrHigh som bytes och idrReserved som 16-bits. Men vi skulle istället kunna ha definierat alla dessa tre som bara

uint32_t idr; dvs 4 bytes och sedan addressera individuella bytes

```
uint8_t  x = *(uint8_t*)&GPIO_E.idr;        // idrLow
uint8_t  y = *((uint8_t*)&GPIO_E.idr  + 1); // idrHigh
uint16_t z = *((uint16_t*)&GPIO_E.idr + 1); // idrReserved
```

```
typedef struct {
    uint32_t    moder;
    uint16_t    otyper;         // +0x4
    uint16_t    otReserved;
    uint32_t    ospeedr;        // +0x8
    uint32_t    pupdr;          // +0xc
    uint8_t     idrLow;         // +0x10
    uint8_t     idrHigh;        // +0x11
    uint16_t    idrReserved;
    uint8_t     odrLow;         // +0x14
    uint8_t     odrHigh;        // +0x15
    uint16_t    odrReserved;
} GPIO;
typedef volatile GPIO* gpioptr;
#define GPIO_E (*((gpioptr) 0x40021000))
```

```
typedef struct _gpio {
    uint32_t    moder;
    uint32_t    otyper;         // +0x4
    uint32_t    ospeedr;        // +0x8
    uint32_t    pupdr;          // +0xc
    uint32_t    idr;            // +0x10
    uint32_t    odr;            // +0x14
} GPIO;
typedef volatile GPIO* gpioptr;
#define GPIO_E (*((gpioptr) 0x40021000))
```

Portadressering med structs

```
typedef struct tag_usart
{
   volatile unsigned short sr;
   volatile unsigned short Unused0;
   volatile unsigned short dr;
   volatile unsigned short Unused1;
   volatile unsigned short brr;
   volatile unsigned short Unused2;
   volatile unsigned short cr1;
   volatile unsigned short Unused3;
   volatile unsigned short cr2;
   volatile unsigned short Unused4;
   volatile unsigned short cr3;
   volatile unsigned short Unused5;
   volatile unsigned short gtpr;
} USART;
#define USART1 ((USART *) 0x40011000)
```

Exempel:

```
while (( (*USART).sr & 0x80) == 0)
    ; // vänta tills klart att skriva
(*USART1).dr = (unsigned short) 'a';
```

Eller med pilnotation:

```
while (( USART->sr & 0x80)==0)
    ;
USART1->dr = (unsigned short) 'a‘;
```

Funktionspekare

Funktionspekare

```c
#include <stdio.h>

int square(int x)
{
    return x*x;
}

int main()
{
    int (*fp)(int);

    fp = square;

    printf("fp(5)=%i \n", fp(5));
    return 0;
}
```

En funktionspekare

fp(5)=25

Funktionspekare

```
int (*fp)(int);
```

Funktionspekarens typ bestäms av:

- Returtyp.
- Antal argument och deras typer.

Funktionspekarens värde är en adress.

Likheter assembler – C

```
        .align   2
delay:
        movs     r3, #255
loop_delay:
        subs     r3, r3, #1
        ands     r3, r3, #255
        bne      loop_delay
        bx       lr

        .align   4
var1:   .SPACE   4
```

Både funktioner och globala variabler har adresser i minnet, men vi använder symboler.

```
int var1;

void delay()
{
    unsigned char tmp = 255;
    do {
        tmp--;
    } while(tmp);
}
```

Structs med funktionspekare

för en mer objektorienterad programmeringsstil

Objektorientering

- Har utvecklats kontinuerligt sedan 50-talet.
- Enormt stort område. Ni lär er i kursen Objektorientering (och fortsättningskurser…)
- Bör vara ert 1:a ryggmärgsval för att designa kod. (Finns emellanåt dock bra andra alternativ.)
- För oss här och nu: ett sätt att strukturera sina funktioner tillsammans med tillhörande data.
- Man vill ha lokala funktioner som hör ihop med strukten. Konceptet kallas **klass** (class). En klass kan innehålla både medlemmar och metoder (=funktioner). C har inte klasser, så vi får simulera detta med structs med funktionspekare för metoderna.

Vi skulle t ex vilja kunna göra såhär:

```
typedef struct {
    int   a;        // en medlem
    void inc() { // metod som inkrementerar a
        a++;
    }
} MyClass;

MyClass v = {0}; // initierar en variabel
v.inc(); // inkrementerar v.a
```

Går ej i C

I C kan vi dock göra såhär:

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

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

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

Objektorientering med C

Förtydligande: Detta är ekvivalent…

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

tMyClass används för att ange en giltig parametertyp för this, eftersom MyClass är odefinierad fram tills sista raden.

… med detta:

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

Talar om för compilern att MyClass är en struct eftersom den inte finns definierad förrän sista raden…

…men behövs redan här

Båda sätten är lika OK.

Hur metodanropet .inc(…) fungerar i C:

```
// MyClass är ett så kallat objekt.
// v1, v2 är två instanser av objektet.
MyClass v1 = {0, incr}, v2 = {1, incr};
// Här anropar vi metoden inc() för v1
// samt v2.
v1.inc(&v1);
v2.inc(&v2);
```

Dessa initieringar gör att v1.a = 0 och v2.a=1 samt att v1.inc och v2.inc pekar på funktionen incr(). Kom ihåg att incr helt enkelt är en symbol för minnesadressen där incr() ligger, samt att v1.inc och v2.inc är pekarvariabler som pekar på denna adress.

Här anropas de funktioner som funktionspekaren v1.inc resp. v2.inc pekar på, och med &v1 resp. &v2 som inparameter. Dvs v1.inc(&v1) resulterar i att incr(&v1) anropas. v2.inc(&v2) resulterar i at incr(&v2) anropas.

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

Anropet incr(&v1) innebär att inparametern this blir lika med &v1 (dvs adressen till v1). Alltså resulterar this->a++ i samma som (&v1)->a++ (vilket är eqvivalent med v1.a++)…

…vilket är precis vad vi vill ska hända när vi gör v1.inc(&v1); ☺

Samma gäller v2.inc(&v2); Dvs det inkrementerar v2.a.

CHALMERS

Objektorientering i C - exempel

Exempel från hemuppgifterna:

```
typedef struct tGameObject{
    // medlemmar
    GfxObject   gfxObj;
    vec2f       pos;
    float       speed;
    // metoder (dvs funktionspekare)
    void (*update) (struct tGameObject* this);
} GameObject;

// update ska peka på någon av dessa funktioner:
void updateShip (GameObject* this)
{
    this->pos += …
    …
}
void updateAlien(GameObject* this)
{
    this->pos += …
    …
}
```

```
GameObject ship, alien;

GameObject* objs[] = {&ship, &alien};

void main()
{
    // initiera funktionspekarna för ship och
    // alien till rätt funktion. (Initiera även
    // övriga structmedlemmar.)
    ship.update = updateShip;
    alien.update = updateAlien;
    …
    // update all objects
    for(int i=0; i<2; i++)
        objs[i]->update(objs[i]);

    …
}
```

Detta resulterar alltså i anropen ship.update(&ship) samt alien.update(&alien), vilket pga update-funktionspekarna resulterar i anropen:
updateShip(&ship) resp. updateAlien(&alien)

Objektorientering i C - exempel

Exempel 5.3 – 5.4 i Arbetsboken:

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

```
// store all objects in global array
OBJECT* obj[] = {&ball, &player};
…
// In some function:
//  - move and draw all objects
for(int i=0; i<2; i++)
{
   obj[i]->move(obj[i]);
   obj[i]->draw(obj[i]);
};
```

```
OBJECT ball =
{
   &ball_geometry, // geometri för en boll
   0,0,            // move direction (x,y)
   1,1,            // position (x,y)
   draw_object,    // draw method
   move_object,    // move method
   set_object_speed   // set-speed method
};
```

```
OBJECT player =
{
   &player_geo, // geometri för en boll
   0,0,         // move direction (x,y)
   10,10,       // position (x,y)
   draw_player, // draw method
   move_player, // move method
   set_player_speed   // set-speed method
};
```

Structs med funktionspekare – påminner om klassmetoder

```
I C:
typedef struct tCourse {
    …
    void (*addStudent)(struct tCourse* crs,
                       char* name);
    …
} Course;

void funcAddStudent(Course* crs, char* name) // some C function
{
   …
}

void main()
{
    Course mop;
    …
    mop.addStudents = funcAddStudent; // set the function pointer to our desired function
    …

    mop.addStudent(&mop, "Per");      // call addStudent() like a class method

    …
}
```

Like a class method!

- **but** needs 4 bytes for the function pointer in the struct . Java/C++ store all the class methods in one separate “ghost” struct.

Structs med funktionspekare – påminner om klassmetoder

```
I C:
typedef struct tCourse {
    char* name;
    float credits;
    int   numberOfParticipants;
    char* students[100];
    void (*addStudent)(struct tCourse* crs, char* name);
} Course;

void funcAddStudent(Course* crs, char* name)
{
   crs->students[crs->numberOfParticipants++] = name;
}

void main()
{
    Course mop;
    mop.name    = "Maskinorienterad Programmering";
    mop.credits = 7.5f;
    mop.numberOfParticipants = 0;
    mop.addStudent = funcAddStudent; // set the function pointer to our desired function

    mop.addStudent(&mop, "Per");
}
```

```
I Java:
public class Course {
  String name;
  float credits;
  int numberOfParticipants;
  String[] students;
  void addStudent(String name);
}

Course mop = new Course();
mop.name = …
mop.credits = 7.5;
mop.addStudent(”Per”);
```

OK, men vi hade ju lika gärna kunnat anropa funcAddStudent(…) här, så vad är poängen med att gå via pmop->addStudent(...)?   Jo...

Övningsuppgift 1

- Gör en struct som innehåller en funktionspekare
- Tilldela funktionspekaren värdet till en motsvarande lämplig funktion (dvs av samma typ).
- Anropa funktionen (dvs motsvarigheten till metodanrop för klass i Java/C++ etc.).

Övningsuppgift 2

Objektorientering – struct med funktionspekare:

```
typedef struct tMyObject {
    int   b;
    … // lägg till funktionspekare till funktion som decrementerar b…
} myObject ;

// … och komplettera nedanstående kod…
myObject var = { … };
var.dec(…); // … så att följande anrop funkar och alltså
            // har samma effekt som var.b--;
```

```
Ledtråd – tidigare exempel:

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

void incr(…)
{
    this->a++;
}
```

Nästa föreläsning:

Mer programstruktur