CHALMERS
UNIVERSITY OF TECHNOLOGY

Machine-Oriented Programming

C-Programming part 3

Pedro Trancoso
ppedro@chalmers.se

Original slides by Ulf Assarsson

Objectives

- Topics:
  - Structs, pointers to structs, array of structs,
  - Port addressing with structs
  - Function pointers,
  - structs with function pointers (object-oriented style)
- HW: v3

Structs: Composite Data Type

A struct:

- Has one or more members (fields).
- Members can be for example of type:
  - base-type
  - int, char, long (as signed/unsigned)
  - float, double
- User defined/composite type (e.g. another struct).
- pointer (even to functions and same struct)

Use of struct

```
#include <stdio.h>

char* coursename = "Machine Oriented Programming";

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

int main()
{
      struct Course mop;

      mop.name = coursename;
      mop.credits = 7.5;
      mop.numberOfParticipants = 110;

      return 0;
}
```

Definition of the structure

Declaration of a variable mop

Access to fields via .-operator

Assembly code to read value of mop.numberOfParticipants?

```
LDR R0, =mop
LDR R1, [R0,#8]
```

Initialization List

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

struct Course c1 = {"MOP", 7.5, 110};
struct Course c2 = {"MOP", 7.5};
```

A `struct` **can** be initialized with an initialization list.
Initiation takes place in the same order as the declarations, but not all members need to be initiated.

Typedef – alias for types

```
typedef unsigned int uint32, uint32_t;

typedef short int int16;

typedef unsigned char *ucharptr;

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

```
typedef int postnr;
typedef int strtnr;
postnr x = 41501;
strtnr y = 3;
x = y; // completely OK

// Note: '*' is not included in the type declaration
for byteptr2
typedef char* byteptr, byteptr2; // byteptr2 wrong!
typedef char *byteptr, *byteptr2; // right
typedef char *byteptr, byte; // right
```

`typedef` simplifies / shortens expressions, which can increase readability.

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

type alias/type name

Structs – Composite data type

Syntax:

optional

```
struct StructName {
  type field1;
  type field2;
  ...
} variable1, variable2 ... ;
```

optional

```
Alternative ways to do the same!
struct {
   int     age;
   char*   name;
} player1, player2;

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

```
Or:
typedef struct tPlayer {// tPlayer can be skipped
   int     age;
   char*   name;
} Player ;
Player player1, player2;
```

Structs – Composite data type

Initialization of structs

Usual commands:

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

//Player is now a type alias for this struct.
Advantage: you do not need to write ”struct Player”

Player player1 = {15, "Ulf"};
Player player2 = {20, "John Doe"};
// or for example
player1.age = 16;
player1.name = "Striker";
```

```
// Structs can contain other structs:
typedef struct {
   int x;
   int y;
} Position;

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

Player player1 = {15, "Striker”, {5, 10}};
// or for example
player1.pos.x = 6;
player1.pos.y = 11;
player1.pos = (Positio

// Incomplete initialization is OK!
Player player1 = {15, "Striker”, {5}};
```

What if you want a Player as a field/member of Player?
(a) Is it possible?
(b) How can we do it?

Assembly code to read value of player1.pos.y?

```
LDR R0, =player1
LDR R1, [R0,#8]
```

Structs – Composite data type

In exercise 5.15 & 5.16 (pg. 108-111) in "Arbetsboken" :

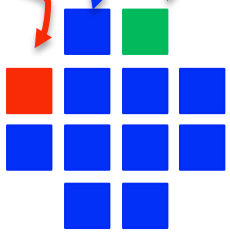
```
typedef struct tPoint{
   unsigned char x;
   unsigned char y;
} POINT;

#define MAX_POINTS 20

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

Assembly code to read value of ball_geometry.px[4].x?

```
@ 12+4*2=20
LDR R0, =player1
LDR R1, [R0,#20]
```

Create and initialize a variable of type 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} // Incomplete initialization
    }       // (12 of 20)
};
```



Pointers to struct / arrow notation

```
Course mop;
Course *pmop; // Pointer to struct

pmop = &mop;

(*pmop).name = …

// Or easier:

pmop->name   = …
```

The arrow notation simplifies the code, as it is common to have pointers for structs

Pointer to struct

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

char* coursename = "Machine Oriented Programming";

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

int main() {
      Course *pmop; // Pointer to struct
      pmop = (Course*)malloc(sizeof(Course));

      //(*pmop).name = coursename;
      pmop->name   = coursename;
      pmop->credits = 7.5;
      pmop->numberOfParticipants = 110;

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

Access to members (fields) via -> operator

```c
// or
Course mop, *pmop;
pmop = &mop;
// and of course without free()
```

In Java:

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

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

Array of structs

```c
#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};
// or simply:
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
```

Port addressing with structs

Instead of:

```
#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))
```

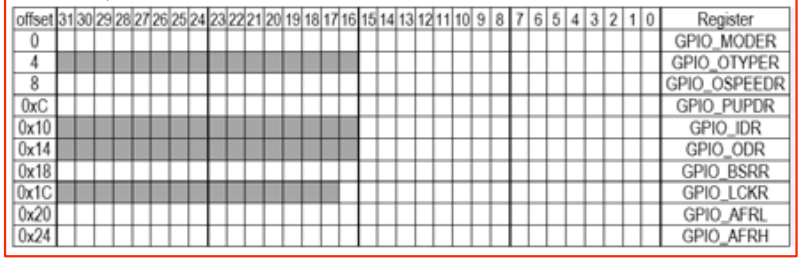
Then we can use structs by writing :

```
typedef struct {
    uint32_t    moder;
    uint16_t    otyper;         // +0x4
    uint16_t    otReserved; //
    uint32_t    ospeedr;        // +0x8
    uint32_t    pupdr;          // +0xc (12)
    uint8_t     idrLow;         // +0x10
    uint8_t     idrHigh;        // +0x11
    uint16_t    idrReserved;
    uint8_t     odrLow;         // +0x14
    uint8_t     odrHigh;        // +0x15
    uint16_t    odrReserved;
} GPIO;
```

GPIO

General Purpose Input Output

GPIO A: 0x40020000
GPIO B: 0x40020400
GPIO C: 0x40020800
GPIO D: 0x40020C00
GPIO E; 0x40021000




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

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

Port addressing – individual bytes

We defined idrLow and idrHigh as bytes and idrReserved as 16-bit. But we could instead have defined all these three as just

uint32_t idr; i.e. 4 bytes and then address individual 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))
```

Port addressing with 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)
```

Example:

```
while (( (*USART).sr & 0x80) == 0)
    ; // wait until it is ok to write
(*USART1).dr = (unsigned short) 'a';
```

Or with the arrow notation:

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

USART

Universal synchronous asynchronous receiver transmitter
USART1: 0x40011000
USART2: 0x40004400

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		USART_SR
4		USART_DR
8		USART_BRR
0xC		USART_CR1
0x10		USART_CR2
0x14		USART_CR3
0x18		USART_GTPR

Function pointer

```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;
}
```

A function pointer

```
fp(5)=25
```

Function pointer

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

A Function pointer type is defined by:

- Return type.
- Number of arguments and their types.

The value of a function pointer is an address.

Question:
What is the definition for a function pointer that points to function foo
char *foo(int *x, int y) {…}

```
LDR R4, =fp
BLX R4
```

```
LDR R4, =foo
BLX R4
```

Structs with function pointers

Programming object-oriented style in a non-object-oriented language!

📖 Arbetsbok pg 109-111

```
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;
```

What are ”draw”, ”clear”, ”move”, ”set_speed”?

Structs with function pointers

Programming object-oriented style in a non-object-oriented language

📖 Arbetsbok pg 109-111

```
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;
```

How do you initialize them?

```
static OBJECT ball = {
   &ball_geometry,
   0, 0,
   1, 1,
   draw_object,
   clear_object,
   move_object,
   set_object_speed
};
```

How do you use them?

```
POBJECT p = &ball;
. . .
p->set_speed(p,4,1);
. . .
p_>move(p);
```

Object-oriented

- Has been developed continuously since the 50's.
- Huge topic. You are taught in the course Object Orientation (and further courses ...)
- Should be your first pick to design code. (However, there are other good alternatives.)
- For us here and now: a way to structure the functions along with associated data.
- You want local functions that are together with the data in the respective structure. The concept is called **class**. A class can contain both **members/variables** (fields) and **methods** (functions). C does not have classes, so we can simulate this using structs with function pointers as the methods.

For example, we would like to do this :

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

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

Not in C!

However, in C, we can do this :

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

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

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

Special attention!

Object-oriented with C

Clarification: This is equivalent ...

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

tMyClass used to enter a valid parameter type for **this**, since MyClass is undefined until the last line.

... to this:

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

Tells the compiler that MyClass is a struct because it is not defined before the last line ...

...but is needed here

Both ways are likely OK.

Object-oriented with C

How the method call .inc(…) works in C:

```
// MyClass is a so called object.
// v1, v2 are two instances of the object.
MyClass v1 = {0, incr}, v2 = {1, incr};
// Here we call method inc() for v1 and v2.
v1.inc(&v1);
v2.inc(&v2);
```

```
// incr() is a regular function (which
function pointer v1.inc and v2.inc point to).
void incr(MyClass* this)
{
    this->a++;
}
```

These initiations make v1.a = 0 and v2.a = 1, and v1.inc and v2.inc point to the incr() function. Remember, incr is simply a symbol for the memory address where incr() is located, and v1.inc and v2.inc are pointer variables that point to that address.

Here are the calls to functions which function pointers v1.inc and v2.inc point to, and using &v1 and &v2 as input parameters. So v1.inc(&v1) results in calling incr(&v1) and v2.inc(&v2) results in calling incr(&v2).

The call incr(&v1) means that input parameter this equals &v1 (i.e. The address to v1).
Thus this->a++ is the same as (&v1)->a++ (which is equivalent to v1.a++)…
… which is exactly what we want to happen when we do v1.inc(&v1);
The same applies to v2.inc(&v2); I.e. it increments v2.a.

Object-oriented in C: Example

Examples of homework:

```c
typedef struct tGameObject{
    // members
    GfxObject   gfxObj;
    vec2f       pos;
    float       speed;
    // methods (i.e. function pointers)
    void (*update) (struct tGameObject* this);
} GameObject;

// update should point to any of these functions:
void updateShip (GameObject* this)
{
    this->pos += …
    …
}
void updateAlien(GameObject* this)
{
    this->pos += …
    …
}
```

```c
GameObject ship, alien;

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

void main()
{
    // initialize the function pointer for ship and
    // alien to the right function (initialize even
    // other struct members)
    ship.update = updateShip;
    alien.update = updateAlien;
    …
    // update all objects
    for(int i=0; i<2; i++)
        objs[i]->update(objs[i]);

    …
}
```

This results in the call of ship.update(&ship) and alien.update(&alien), which due to the update-function-pointer results in calling: updateShip(&ship) and updateAlien(&alien)

Object-oriented in C: Example

Example 34 in “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,        // geometry for a ball
    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,           // geometry for a ball
    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 with function pointers:

revision of class methods

```
In 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 with function pointers:

revision of class methods

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

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

void main() {
      Course mop;
      mop.name    = "Maskinorienterad Programmering";
      mop.credits = 7.5f;
      mop.numStudents = 0;
      mop.addStudent = funcAddStudent; // set the function pointer to
      mop.addStudent(&mop, "Per");
}
```

In Java:

```
public class Course {
  String name;
  float credits;
  int numStudents;
  String[] students;
  void addStudent(String name) {
    students[numStudents++]=name;
  }
}

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

OK, but we could as well have been able to call funcAddStudent(…) here, So what's the point of going through pmop->addStudent(...)?

Function pointers

Why?

```
int(*print)(const char*,…);

print = printf;
or
print = popupWindow;

Structs with function pointers – why?
```

Object-oriented. Methods:

```
void updateObjects(struct Objects objs[], int numObjs)
{
   for (int i=0; i<numObjs; i++)
      objs[i].update(&objs[i], …);
}
```

Function pointers for the ”ninjas”

Define max and min functions for int and char.

Use an array of function pointers for max and min.

```
#include <stdio.h>

#define T_INT   0
#define T_CHAR  1

void maxInt( void *x, void *y ) {
  if( *((int*)y) > *((int*)x) ) *((int*)x) = *((int*)y);
}
void minInt( void *x, void *y ) {
  if( *((int*)y) < *((int*)x) ) *((int*)x) = *((int*)y);
}

void maxChar( void *x, void *y ) {
  if( *((char*)y) > *((char*)x) ) *((char*)x) = *((char*)y);
}
void minChar( void *x, void *y ) {
  if( *((char*)y) < *((char*)x) ) *((char*)x) = *((char*)y);
}

void processArray( void (*func)(void*,void*), void *res, void *array,
unsigned int size, char type ) {
  for( int i = 0; i < size; i++ )
    if( type == T_INT )
      (func)(res, (void*)(((int*)(array))+i));
    else
      (func)(res, (void*)(((char*)(array))+i));
}
```

```
void (*max[])(void*,void*) = { maxInt, maxChar };
void (*min[])(void*,void*) = { minInt, minChar };



int main() {
  int x[] = { 2, 5, 6, 111, -10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  int iout;

  iout = x[0];
  processArray( max[T_INT], &iout, x, 15, T_INT );
  printf( "iout = %d\n", iout);


  char y[] = { 'd', 'e', 'j', 'q', 'i', 'b' };
  char cout;

  cout = y[0];
  processArray( max[T_CHAR], &cout, y, 6, T_CHAR );
  printf( "cout = %c\n", cout);

}
```