CHALMERS
UNIVERSITY OF TECHNOLOGY

Machine-Oriented Programming

C-Programming part 2

Pedro Trancoso
ppedro@chalmers.se

Original slides by Ulf Assarsson

Contents

- Topics:
  - Pointers
  - Absolut addressing (ports)
  - typedef, volatile, #define
  - Arrays of pointers, arrays of arrays
- Exercises:
  - v2

Previous C Lecture

- C-syntax
- Program structure, compiling, linking
- Bitwise operations

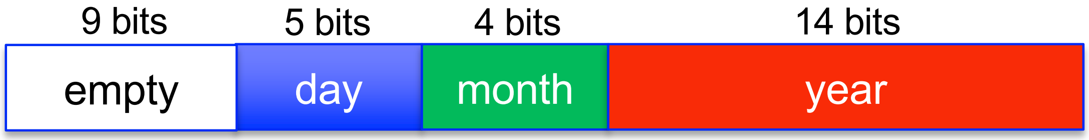
Quick Review:

- 0x (prefix for hexadecimal), 0b (prefix for binary)
- a << n (shift n bits of a to the left and fill with 0 bits coming from the right)
- a >> n (shift n bits of a to the right and fill with 0 bits coming from the left)

Bitwise operations: Assignment

Packing different values into a single variable:

- Pack and Unpack a date (DAY/MONTH/YEAR) into a word (integer) variable

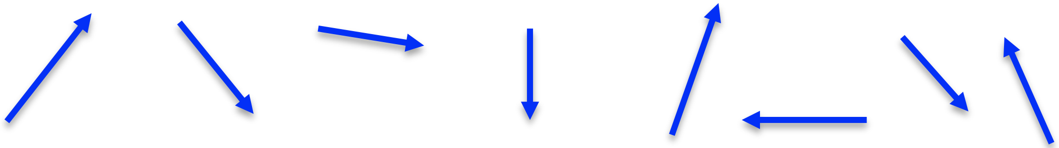


9 bits	5 bits	4 bits	14 bits
empty	day	month	year

```
#define DAYMASK      0xFF83FFFF
#define MONTHMASK    0xFFFC3FFF
#define YEARMASK     0xFFFFC000
#define _5BITMASK    0x0000001F
#define _4BITMASK    0x0000000F
#define _14BITMASK   0x00003FFF

int date = 0;

void setDay(int day) {
    day = day << 18;
    date = (date & DAYMASK) | day;
}

void setMonth(int month) {
    month = month << 14;
    date = (date & MONTHMASK) | month;
}

void setYear(int year) {
    date = (date & YEARMASK) | year;
}
```

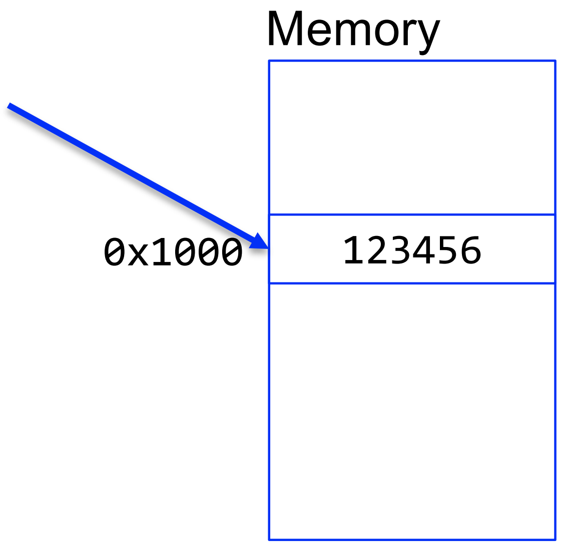
```
int getDay(void) {
    int day = date;
    return (day >> 18) & _5BITMASK;
}

int getMonth(void) {
    int month = date;
    return (month >> 14) & _4BITMASK;
}

int getYear(void) {
    int year = date;
    return year & _14BITMASK;
}

int main(int argc, char **argv) {
    return 0;
}
```

Pointers

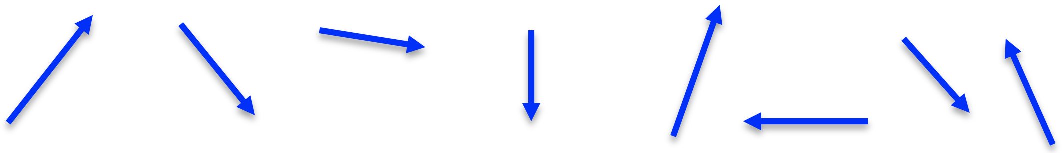


- A pointer is a variable that holds a memory address of a value (e.g., variable or port), instead of holding the actual value itself.

Pointer to value ”123456”, i.e. location of value ”123456” in memory, i.e. its address! (0x1000)

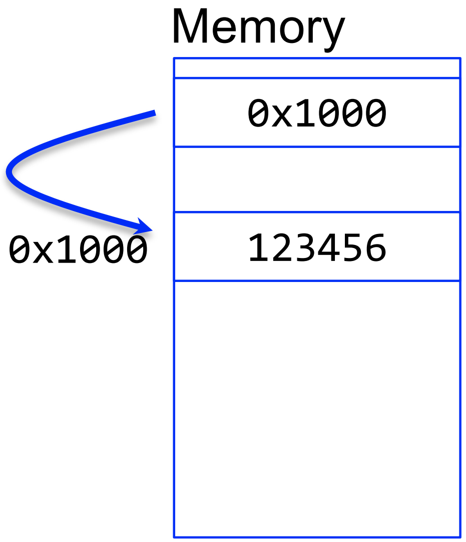


Pointers



- **A pointer is a variable that holds a memory address of a value (e.g., variable or port), instead of holding the actual value itself.**

Pointer to value ”123456”, i.e. location of value ”123456” in memory, i.e. its address! (0x1000)



A pointer is essentially a variable that holds a **memory address** of a value (variable or port), instead of holding the actual value itself.

Why pointers?

- **Allows to refer to an object or variable, without having to create a copy**

```
Example 1:

char person1[] = "Elsa";
char person2[] = "Alice";
char person3[] = "Maja";
…
char* winner = person2;
char* winner = &(person2[0]);
```

Are both the same?
What about:
winner=&(person2[2])

```
Example 2:

int salaryLevel1 = 1000;
int salaryLevel2 = 2000;
int salaryLevel3 = 3000;
…
int* minSalary = &salaryLevel3;
…
minSalary = &salaryLevel1;
…
X = minSalary + 1000;
Y = *minSalary + 1000;
```

Are both the same?

winner points to person2.

Pointers

”&a” – The address of a
”*a” – The contents in address a

1. The pointer’s value is an address (&).
2. The pointer’s type tells how one interprets the bits in the content.
3. ”*” is used to read (derefer) the content of the address.

```
int salaryLevel1 = 1000;
int salaryLevel2 = 2000;
int salaryLevel3 = 3000;

int* minSalary = &salaryLevel3; // == 0x20030108
```

type

value is an address

```
minSalary is 0x20030108
*minSalary is 3000.

printf(“min salary = %d kr”, *minSalary);
```

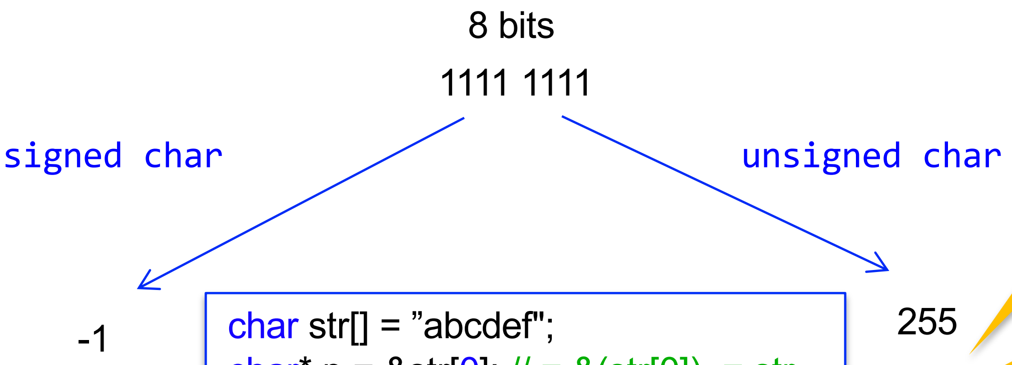
```
min salary = 3000 kr
```

	0x20030108	minSalary
…	…	
0x20030108	3000	salaryLevel3
0x20030104	2000	salaryLevel2
0x20030100	1000	salaryLevel1
…	…	
0x00000001		
0x00000000		

Increasing Addresses

Pointers: dereference “*”

- When we dereference a pointer we get the object that is stored in the corresponding address
  - The number of bytes we read depends on the type
  - The interpretation of the bits depends on the type



8 bits

1111 1111

signed char

unsigned char

-1

255

```
char str[] = ”abcdef";
char* p = &str[0]; // = &(str[0]), = str
char s = *p;
```

What is the output?

```
char *x = &str[1];
printf(”%s\n”, x);
```

What is the output?

```
char *p = &str[0];
printf(”%s\n”, (++p));
```

What is the output?

```
int *p = (int*)&str[0];
printf(”%s\n”, (char*)(++p));
```

Pointers: Operators & *

```c
#include <stdio.h>

int main() {
    char a, b, *p;
    a = 'v';

    b = a;
    p = &a;

    printf("b = %c, p = 0x%p (%c) \n", b, p, *p);

    a = 'k';
    printf("b = %c, p = 0x%p (%c) \n", b, p, *p);
}
```

Pointer declaration

Address of ...

Dereferering

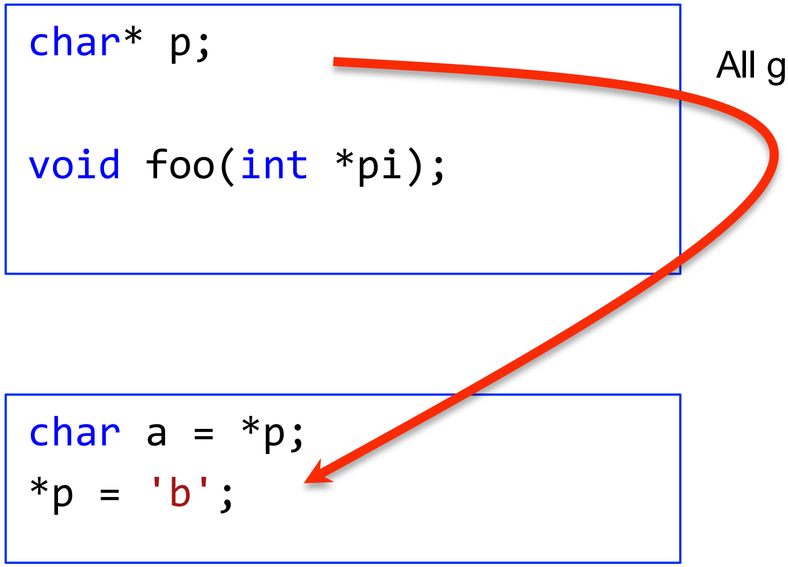
?

```
Output:
b = v, p = 0x0027F7C3 (v)
b = v, p = 0x0027F7C3 (k)
```

Meaning of "*"

- In declarations:
  - Pointer type

- As operator
  - dereferens



```
char* p;

void foo(int *pi);
```

```
char a = *p;
*p = 'b';
```

All good?

Pointers: Summary

If a’s value is an address, *a is the content of that address.

&a Address of variable a. The memory address where a is stored.

a Variable’s value (e.g. int, float or an address if a is a pointer variable)

*a The variable a points to. Here a’s value must be a valid address (e.g. pointer to another variable or port) and a must be of type pointer. “*a” is used to get the value for the variable/port.

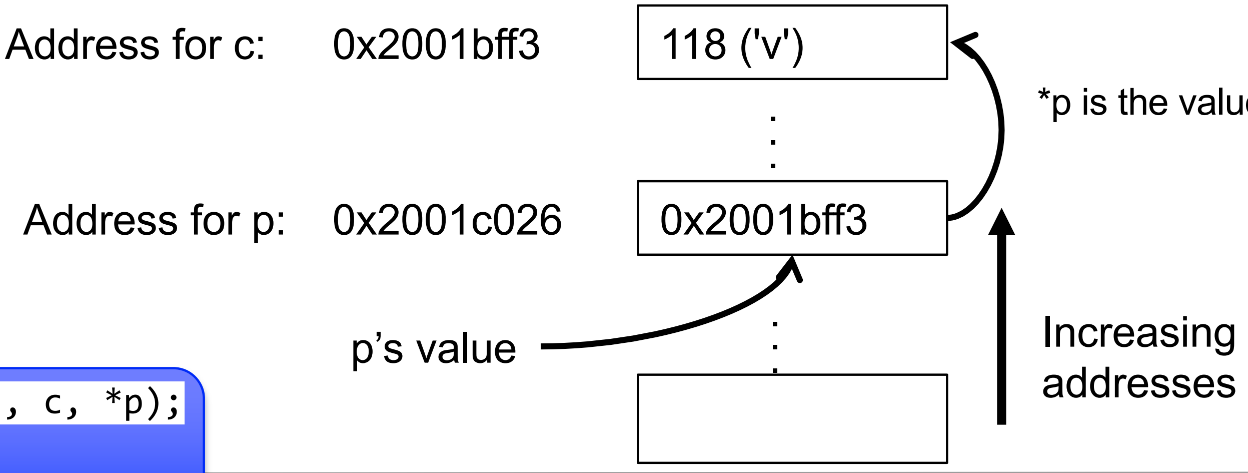
Example:

```
char  c = 'v';
…
char* p = &c;
```

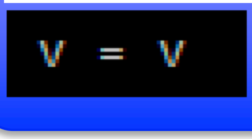
&p is 0x2001c026

p is 0x2001bff3

*p is ‘v’



```
printf("%c = %c", c, *p);
```



```
v = v
```

Pointers: Example

```
char person1[] = "Elsa";
char person2[] = "Alice";
char person3[] = "Maja";
…
char* winner = &person2[0]; // == 0x20030200
```

How many bytes is the content?

type

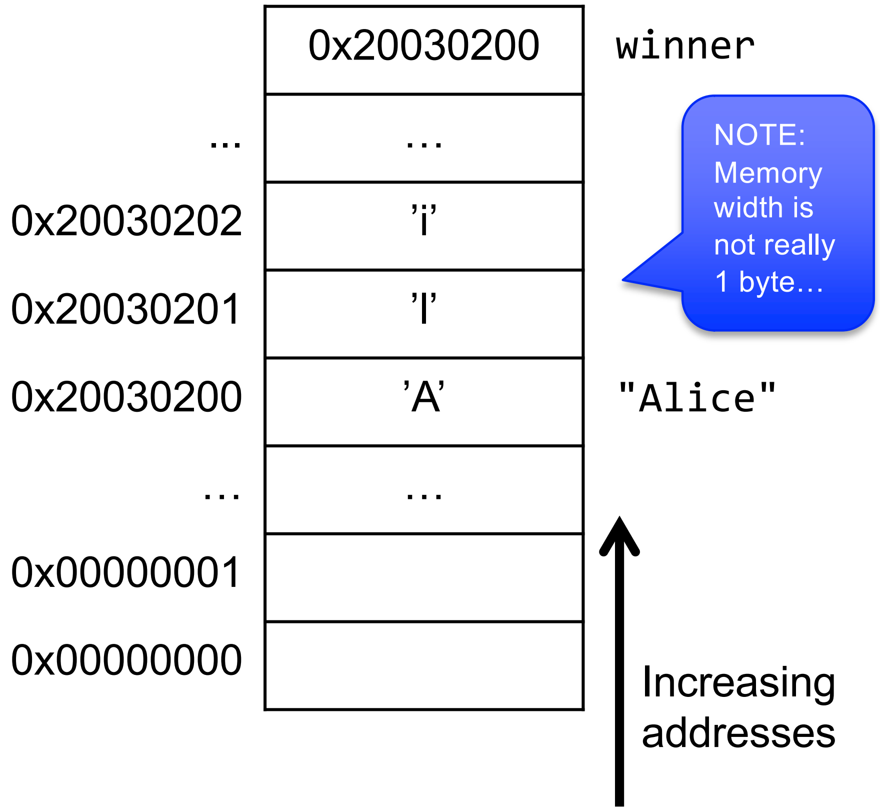
Value is an address

```
winner is 0x20030200
*winner is "Alice".
```

```
printf(”%s”, winner);
```

Fill-in-the-gaps!

```
char *chp = winner;
while( … )
   printf(”%c”, … );
```



	0x20030200	winner
…	…	
0x20030202	’i’	
0x20030201	’l’	
0x20030200	’A’	"Alice"
…	…	
0x00000001		
0x00000000		

NOTE: Memory width is not really 1 byte…

Increasing addresses

Pointers: More pointers

```
int  a[] = {2,3,4,10,8,9};
int  *pa = &a[0];

short int b[] = {2,3,4,10,8,9};
short int *pb = b;

float     c[] = {1.5f, 3.4f, 5.4f, 10.2f, 8.3f, 2.9f};
float     *pc = &c[3];
```

Pointers to string:

```
char course[] = "Machine-Oriented Programming";
char *pCourse = course;
```

“course” is a standard writable array on the stack or in the program's data segment.

Or directly as in:

```
char *pCourse = ”Programming of Embedded Systems";
```

But here the C compiler places the string in read-only string memory in the program's data segment

Pointers

What is the value of *p if p is of type int*?

`t *p;`	p declared of type “pointer to type t“
`p = 0;`	p becomes a null pointer (pointer to nothing!)
`p = &v;`	p is assigned the address of variable v
`*p`	means “content of what p points to”
`p1 = p2;`	p1 will point to the same pointed by p2
`*p1 = *p2;`	content of what is pointed by p1 becomes the same as the content of what p2 points to.

Why pointers?

- Write to/Read from ports
- (faster indexing in arrays)
- Use copies of input parameters
- Change the input parameters…

```
#include <stdio.h>

void inc(int x, char y)
{
    x++;
    y++;
}
```

Arguments are ”pass-by value” in C.

```
int var1 = 2;
char var2 = 7;
inc(var1, var2);
```

var1 and var2 have still values 2 and 7 after the function call

```
#include <stdio.h>

void inc(int *x, char *y)
{
    (*x)++;
    (*y)++;
}
```

Arguments are ”pass-by value” in C.

```
int var1 = 2;
char var2 = 7;
inc(&var1, &var2);
```

var1 and var2 have now values 3 and 8 after the function call

Pointer arithmetic

```
char *course = ”Machine-Oriented Programming";

*course;        // 'M’
*(course+2);    // ‘c’
course++;       // course now points to 'a’
course += 4;    // course now points to ’i'
```

What is the result of:

1. printf(“%c\n”, *course);
2. printf(“%s\n”, course);

p is increased by (n * size_of_type)

Assume p=0x00000000, what is the value of p after p++?

1. In case char *p
2. In case int *p

```
int a[] = {2,3,4,10,8,9};

int *p = a;      // p == &(a[0])
p++;             // p == &(a[1])

int *p3 = a + 3;// p == &(a[3])
```

Pointer Examples (…think…)

```
char name[] = ”Machine Oriented Programming”;
char *p2c;
int  *p2i;

p2c = name;
p2i = (int*)name;
printf( ”%s - %s\n”, (char*)p2c, (char*)p2i);

p2c += 3;
p2i += 3;
printf( ”%s - %s\n”, (char*)p2c, (char*)p2i);
```

Machine Oriented Programming - Machine Oriented Programming

hine Oriented Programming - nted Programming

Pointers for absolute addressing

- As a port ”identifier” we can have an absolute address (e.g. 0x40011004).

Absolute addressing

```
0x40011000                       // an hexadecimal number
(unsigned char*) 0x40011000      // an unsigned char pointer that points to address 0x40011004
*((unsigned char*) 0x40011000)  // dereferens of the pointer

// Read from 0x40011000
unsigned char value = *((unsigned char*) 0x40011000);

// Write to 0x40011004
*((unsigned char*) 0x40011004) = value;
```

But… we need to add <u>volatile</u> if we have optimization flags... !

User defined types with typedef

Preprocessor does all the work!

```
#define INPORT *((unsigned char*) 0x40011000)
value = INPORT;
```

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

INPORT_ADDR
(port8ptr)INPORT_ADDR
INPORT

// read from 0x40011000
value = INPORT;
```

```
Evaluates to:



0x40011000
(unsigned char*) 0x40011000
*((unsigned char*) 0x40011000)

// read from 0x40011000
value = *((unsigned char*) 0x40011000);
```

`typedef` simplifies / shortens expressions, to increase readability.

```
typedef unsigned char* port8ptr;
```

type alias/type name

Volatile qualifier

```
char * inport = (char*) 0x40011000;

void foo(){

     while(*inport != 0)
     {
          // ...
     }
}
```

A compiler that optimizes may only read once (or not at all if we never write to the address from the program).

Volatile qualifier

```c
volatile char * inport = (char*) 0x40011000;

void foo(){

    while(*inport != 0)
    {
        // ...
    }
}
```

```c
volatile char * utport = (char*) 0x40011000;

void f2()
{
    *utport = 0;
    …
    *utport = 1;
    …
    *utport = 2;
}
```

`volatile` prevents some optimizations (which is good and necessary!), i.e. ***indicates that the compiler must assume that the content of the address can be changed from outside***.

The previous example, now corrected with volatile:

```c
unsigned char value = *((volatile unsigned char*) 0x40011000); // read from 0x40011004

*((volatile unsigned char*) 0x40011004) = value; // write to 0x40011004
```

Summary for ports

In-port:

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

// read from 0x40011000
value = INPORT;
```

Out-port:

```
typedef volatile unsigned char* port8ptr;
#define UTPORT_ADDR 0x40011004
#define UTPORT *((port8ptr)UTPORT_ADDR)

// write to 0x40011004
UTPORT = value;
```

Pointers and Arrays

Number of bytes with sizeof()

```
#include <stdio.h>

char* s1 = "Emilia";
char s2[] = "Emilia";

int main()
{
    printf("sizeof(char):  %d \n", sizeof(char) );
    printf("sizeof(char*): %d \n", sizeof(char*) );
    printf("sizeof(s1):    %d \n", sizeof(s1) );
    printf("sizeof(s2):    %d \n", sizeof(s2) );

    return 0;
}
```

```
sizeof(char):     1
sizeof(char*):    4
sizeof(s1):       4
sizeof(s2):       7
```

Sizeof evaluated at compile-time. One (of few) exceptions where arrays and pointers are different.

It is actually a “string” not an “array”

Indexing: Same for array / pointers

x[y] is translated to *(x + y) and is thus a way to derive a pointer.
Indexing is the same for pointers as for the array.
So are arrays pointers? No…

```
#include <stdio.h>

char* s1 = "Emilia";
char s2[] = "Emilia";

int main()
{
     // tre ekvivalenta sätt att dereferera en pekare
     printf("'l’ in Emilia (version 1): %c \n",    *(s1+3));
     printf("'l’ in Emilia (version 2): %c \n",    s1[3]);
     printf("'l’ in Emilia (version 3): %c \n",    ~~3[s1]);~~

     // tre ekvivalenta sätt att indexera en array
     printf("'l’ in Emilia (version 1): %c \n",    *(s2+3));
     printf("'l’ in Emilia (version 2): %c \n",    s2[3]);
     printf("'l’ in Emilia (version 3): %c \n",    ~~3[s2]);~~

     return 0;
}
```

Arrays vs Pointers: Similarities and Differences

```
char* s1 = "Emilia";
char s2[] = "Emilia";
```

- Both have and address and a type.
  - `char s2[] = "Emilia";`
    - `sizeof(s2) = 7`
  - `char* s1 = "Emilia";`
    - `sizeof(s1) = sizeof(char*) = 4`

```
s1++; // is allowed
s2++; // is NOT allowed
```

- Indexing has the same result.
  - s1[0] → ’E’
  - s2[0] → ’E’
  - *s1 → ’E’
  - *s2 → ’E’ (because s2 is an address, we can dereference it just like a pointer)

Arrays vs Pointers: Similarities and Differences

```
char* s1   = "Emilia";
char  s2[] = "Emilia";
```

	s2	s1
Type:	Array	Pointer variable

Addressing:	**&s2 is not possible -** s2 is just a symbol
	s2 = symbol = array’s start address.
	s2 = &(s2[0])
	s2[0] ≡ *s2 → 'E'
		&s1 = address for variable s1.
		s1 = s1’s value = string’s start address.
		s1 = &(s1[0])
		s1[0] ≡ *s1 → 'E'

Pointer arithmetic:	**s2++ is not possible** (s2+1)[0] is OK	s1++ is OK (s1+1)[0] is OK
Size of type:	sizeof(s2) = 7 bytes	sizeof(s1) = sizeof(char*) = 4 bytes

s2 is a symbol (not a variable) for an address which is known at compile time.
Because s2 is an address we can dereference it exactly as a pointer: *s2 → ’E’.

Indexing: More Examples

```
#include <stdio.h>

char * s1 = "Emilia";  // s1 is a pointer. Variable s1 is a variable which can be changed,
                       // and at start the value is assigned the address to 'E’
char s2[] = "Emilia";  // s2 is an array. The value of symbol s2 is known at compile time.
                       // Symbol s2 is constant, not like a variable which value can be changed.
                       // The value of s2 is an address to 'E’.
int main()
{
   // three equivalent ways to dereference a pointer
   printf("'l’ in Emilia (version 1): %c \n",  *(s1+3)     );
   printf("'l’ in Emilia (version 2): %c \n",  s1[3]       );
   printf("'l’ in Emilia (version 3): %c \n",  *(s2+3)     );
   printf("'l’ in Emilia (version 3): %c \n",  (s2+3)[0]   );

   char a[] = "hej";
   (a+1)[0] = 'o';
   char* p = a;
   p = "bye"; // works! String ”bye" is allocated at compile time as a read-only s

   char b[10] = "hej"; // b becomes 10 elements.
   // b = "då"; // here we try to change b's value, but it does not go through "..." synta
   b[0] = 'd';  // OK
   b[1] = 'å';   // OK
   b[2] = 0;    // OK OR b[2] = ’\0’

   return 0;
```

```
char b[10] = "hej"; // b becomes 10 elements.
b[4] = 'd';
b[5] = 'a';
b[6] = 0;

printf("b=%s\n", b);
```

Arrays as function parameters become pointers

```
void foo(int i[]);
```

```
void foo(int *i);
```

[ ] – the notation exists but it means pointer!

Avoids the entire array to be copied. *Length not always known at compile time*. The address of the array is added to the stack and accessed via the stack variable i.

(A struct is copied and placed on the stack).

```
int sumElements(int *a, int l)
{
    int sum = 0;
    for (int i=0; i<l; i++) {
      sum += a[i];
    }
    return sum;
}
…
int array[] = {5,4,3,2,1};
int x;
x = sumElements(array, 5);
```

Array of pointers

```
#include <stdio.h>

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

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

    return 0;
}
```

```
Droopy, Emilia, Emil

sizeof(manyName) = 12; // 3*sizeof(char*) = 3*4 = 12
```

Array of arrays

```
#include <stdio.h>

char shortName[][4] = {"Tor", "Ulf", "Per”, “Ian” };

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

    return 0;
}
```

```
Per, Ulf, Tor

sizeof(shortName) = …
```

Array of arrays

```
#include <stdio.h>

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

int main()
{
    int i,j;
    for( i=0; i<3; i++) {
        printf("arrayOfArray[%d] = ", i);
        for ( j=0; j<4; j++)
            printf("%d ", arrayOfArrays[i][j]);
        printf("\n");
    }

    return 0;
}
```

arrayOfArrays[i][j] = arrayOfArrays+i*4+j

Exercises

1. Create a port to an int located at the address 0x40004000.

2. Create a pointer to a string (“hej”) which is in read-only string-letter memory in the data segment.

3. Create a pointer to a string (“hej”) located on the stack.

4. Use typedef to create a new type byteptr as pointer to unsigned byte.

5. What does volatile do?

1.
```
typedef volatile int* port8ptr;
#define PORT_ADDR 0x40004000
#define PORT *((port8ptr)PORT_ADDR);
```

2.
```
char *p = “hej”;
```

3.
```
void fkn()
{
    char s[] = “hej”; // in the stack
    char* p = s;
}
```

4.
```
typedef unsigned char *byteptr;
```

5. Reading/writing of the volatile variable is not optimized. Volatile therefore is necessary for ports.

Next (C) Lecture:

- **Structs**
- **Function pointers**

```
struct abc {
   int a;
   char b;
   short c;
};

struct abc x;

x.a = 2345678;
x.b = ’f’;
x.c = 572;
```

```
2345678, f, 572
```

```
union abc {
   int a;
   char b;
   short c;
};

union abc x;

x.a = 2345678;
x.b = ’f’;
x.c = 572;
```

```
2294332, <, 572
```