CHALMERS
UNIVERSITY OF TECHNOLOGY

Machine-Oriented Programming

C-Programming part 2

Pedro Trancoso
ppedro@chalmers.se

Original slides by Ulf Assarsson

Contents

- Scope / Visibility
- Types2: Arrays, strings, structures, typedef
- Pointers
- Pointer arithmetic
- Absolute addressing
- Volatile
- Ports

Visibility / Scope

- Global visibility (global scope)
- File visibility (file scope)
- Local visibility (e.g. function scope)

Visibility

```
#include <stdio.h>

char x;

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

char y;
```

Visibility at the function level

```
#include <stdio.h>

char x;

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

Visibility at the function level

```
#include <stdio.h>

char x;

int foo()
{
   int x = 4;
   return x;
}
```

Which visibility has the highest priority?

```
#include <stdio.h>

int x;

int foo(int x) {
    if( x == 0 ){
        int x = 4;
        return x;
    }
    return x;
}

int main() {
    x = 1;
    x = foo(0);
    printf("x is %d\n", x);
    return 0;
}
```

What is the output (value of x)?

Arrays

```
int   a[] = {3, 2, 1, 0};
int   b[5];
float c[6] = {2.0f, 1.0f};

int main()
{
    a[0] = 5;
    b[4] = a[2];
    c[3] = 3.0f;
    return 0;
}
```

Strings

Strings are null-terminated character arrays

```
#include <stdio.h>

char name1[] = {'E', 'm', 'i', 'l', '\0'};
char name2[] = "Emilia";
char name3[];

int main()
{
    printf("name1: %s \n", name1);
    printf("name2: %s \n", name2);
    printf("sizeof (name2): %d \n", sizeof(name2));

    return 0;
}
```

?

Output:

```
name1: Emil
name2: Emilia
sizeof(name2): 7
```

7 what?

Exercise:

Have an array where position 0 you have the total number of ‘a’ in a text, in position 1 you have the total number of ‘b’, etc.

```
int   countChars[100];

int main() {
    ...
    if( ch == ‘a’ )
        countChars[0]++;
    else if( ch == ‘b’ )
        countChars[1]++;
    else if( ch == ‘c’ )
    ...
   return 0;
}
```

Better? Single line for all cases?

```
int   countChars[100];

int main() {
    ...
    countChars[ch-’a’]++;
    ...
   return 0;
}
```

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,#12]
```

Structs – Composite data type

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

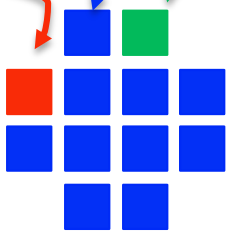
```
typedef struct tPoint{
   unsigned char x;
   unsigned char y;
} POINT;

#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?

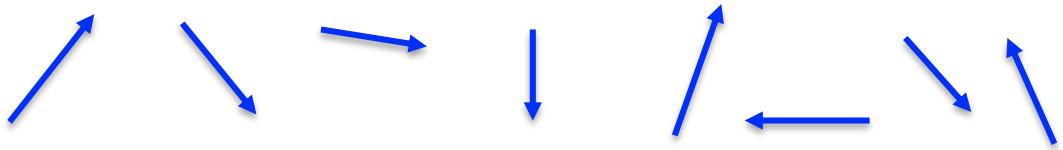
```
@ 12+4*2=20
LDR R0, =player1
LDR R1, [R0,#20]
```

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)
};
```

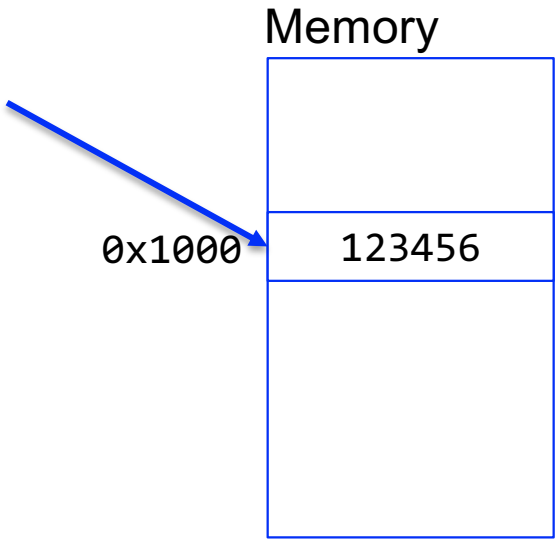


Pointers

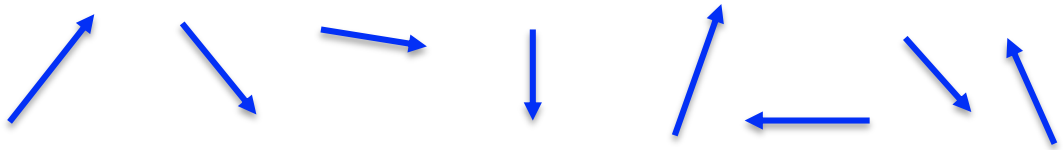


- **A pointer is a variable that holds a memory address of a value (e.g., variable or port), instead of holding the actual value itself.**

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

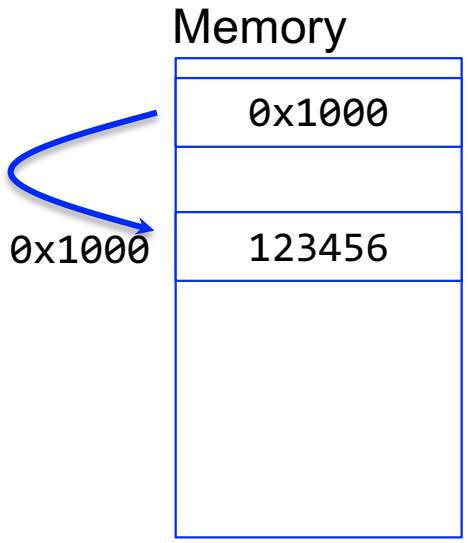


Pointers



- **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])

winner points to person2.

```
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?

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.

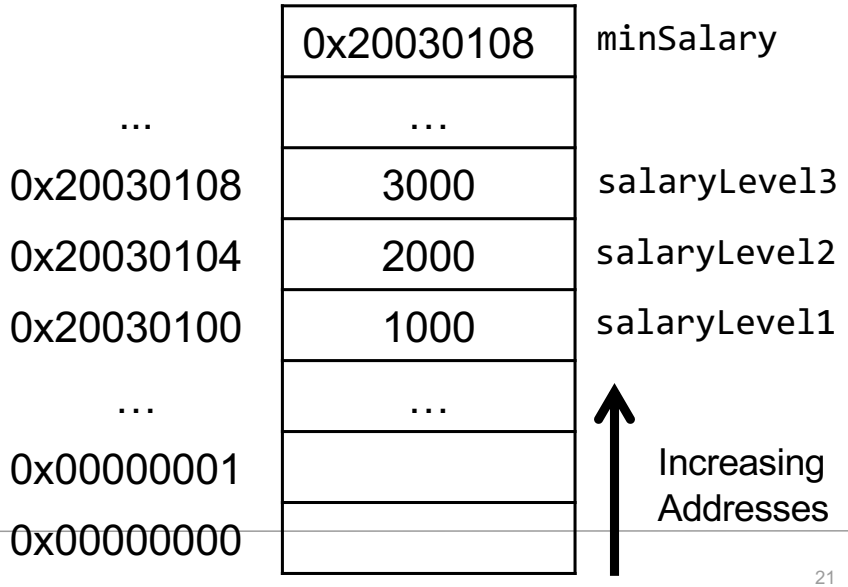
```
int salaryLevel1 = 1000;
int salaryLevel2 = 2000;
int salaryLevel3 = 3000;

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

type

value is an address

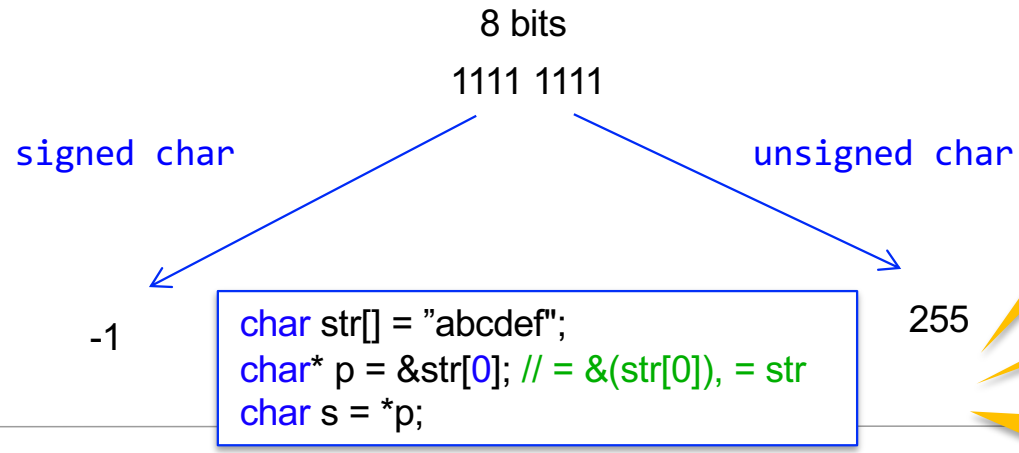
```
minSalary is 0x20030108
*minSalary is 3000.

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

min salary = 3000 kr



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



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 & *

```
#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 "*"

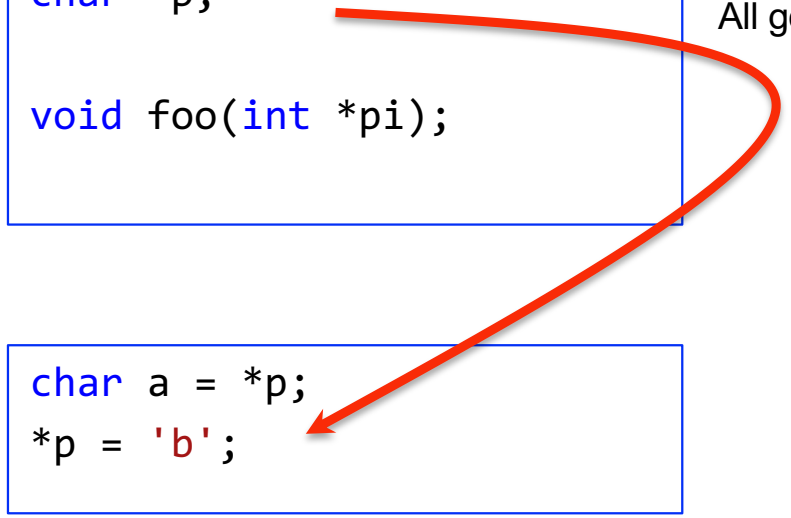
- In declarations:
  - Pointer type

```
char* p;

void foo(int *pi);
```

All good?

- As operator
  - dereferens

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



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.

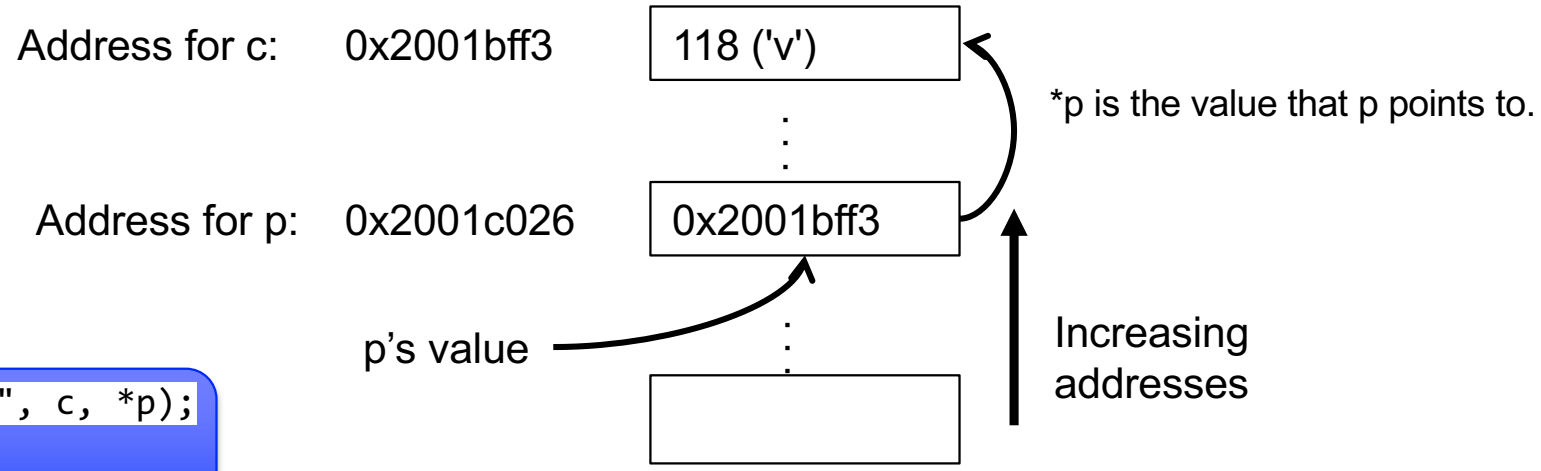
Example:

```
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
```

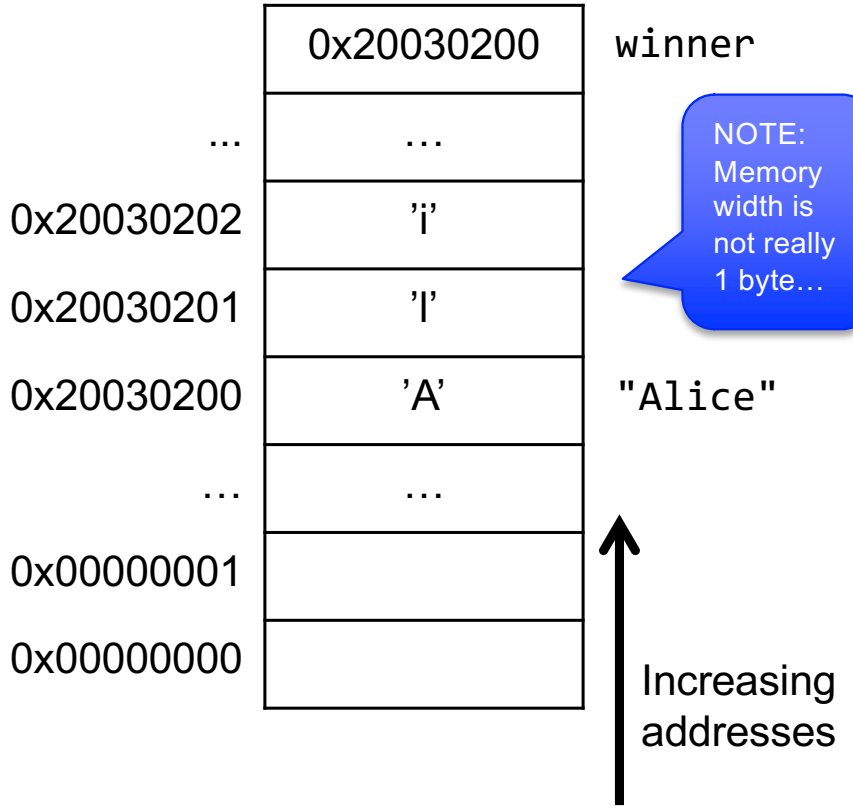
How many bytes is the content?

type

Value is an address

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

```
printf(”%s”, winner);

Fill-in-the-gaps!

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



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.

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

Why pointers?

```c
#include <stdio.h>

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

Arguments are ''pass-by value'' in C.

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

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

```c
#include <stdio.h>

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

Arguments are ''pass-by value'' in C.

```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 volatile 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

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

void foo(){

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

```
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:

```
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;
```

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.