

Computer Architecture (Practical Class)

Pointers in C

Luís Nogueira Raquel Faria

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

`{l,n,r}@isep.ipp.pt`

2020/2021

- A program sees memory as one large array containing bytes
- We do not use the term "index" when referring to a memory location. We use the term *address*
- When you declare a variable, you are reserving one or more continuous bytes of memory
- Each declared variable has an *address*, which indicates where the variable data starts in the memory
- In IA-32 addresses are 32 bits long, thus can vary between 0 and 4GB

- C has special variables, called *pointers*, that are used to store memory addresses
- Pointers are declared like normal variables, with a type associated to it (we will see how this is used later)

Pointers always have the same size

- The size of an address of the underlying architecture (32 bits = 4 bytes in IA32)
- Pointers allow direct access to memory, making it possible to change the values in the memory addresses stored in the pointers
- Some tasks are easier/more efficient to implement when using pointers, and some (such as dynamic memory allocation) are only possible using pointers

- Just like any variable in C, a pointer must be declared before being used
- A pointer is declared using type `*` before the variable identifier:

```
int *ptr1; // declares a pointer to an integer
char *ptr2; // declares a pointer to a char
```

- To obtain the address of a variable, use `'&'` before its identifier:

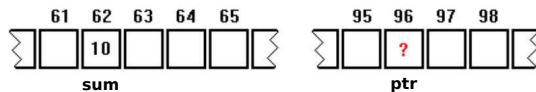
```
int x;
char c;
ptr1 = &x; // store the address of x in ptr1
ptr2 = &c; // store the address of c in ptr2
```

- The *dereference operator* `'*'` accesses the value at a memory address:

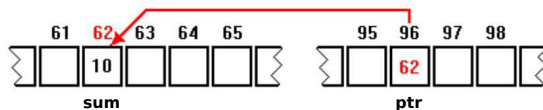
```
*ptr1 = 10; // assign 10 to the value pointed by ptr1
*ptr2 = 'X'; // assign 'X' to the value pointed by ptr2
```

Content of a variable and its address (1/2)

```
char sum;  
char *ptr;  
sum = 10;
```

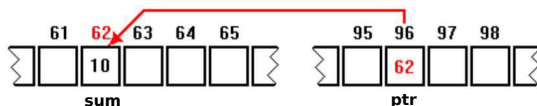


```
ptr = &sum;
```

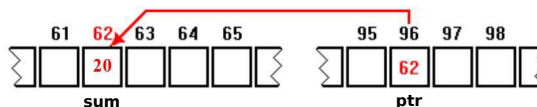


Content of a variable and its address (2/2)

- To access the contents of a memory address in a pointer, use the * operator:



`*ptr = 20;`



Pointers should always be initialized to a valid address before being used

- Using an uninitialized pointer has undefined behaviour!

- Wrong way¹:

```
int x;  
int *ptr; // uninitialized pointer  
  
*ptr=22: // this is a very bad idea!
```

- Correct way:

```
int x;  
int *ptr;  
  
ptr=&x; // now is initialized to the address of x  
*ptr=22: // assign 22 to the memory addressed by ptr (x)
```

¹This code will, most likely, result in a segmentation fault. A segmentation fault occurs when a program tries to access an invalid memory address. The operating system detects this and terminates the program.

- We have said that the memory is a large array of bytes... So, how do we store data larger than 1 byte?
- Easy: we divide the data into bytes and store it! But, this means we have two ways of storing data in memory:
 - Big Endian
 - Store the most significant byte in the smallest address.
 - Adopted in platforms by Sun, PPC Mac, transferring data on the Internet
 - Little Endian
 - Store the least significant byte in the smallest address.
 - Adopted by x86, ARM

Little Endian example

- Let us assume we want to store the number 305419896 (0x12345678):

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
0x12								0x34								0x56								0x78							

- Byte by byte

7	6	5	4	3	2	1	0	
0x78								} Address x
0x56								} Address $x + 1$
0x34								} Address $x + 2$
0x12								} Address $x + 3$

Big Endian example

- Let us assume we want to store the number 305419896 (0x12345678):

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
0x12								0x34								0x56								0x78							

- Byte by byte

7	6	5	4	3	2	1	0	
0x12								} Address x
0x34								} Address $x + 1$
0x56								} Address $x + 2$
0x78								} Address $x + 3$

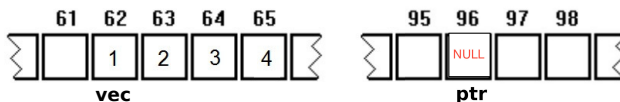
Listing 1: data-rep.c (<http://codepad.org/leCxR8yX>)

```
#include <stdio.h>
int main(){
    unsigned int i = 0xFFAAEEBB;
    int b = 0x1234;
    char c = 'A';
    unsigned int x = 0x12345678;
    // %p: Pointer address format specifier
    printf("i em %p\n",&i);
    printf("b em %p\n",&b);
    printf("c em %p\n",&c);
    printf("x em %p\n",&x);
    return 0;
}
```

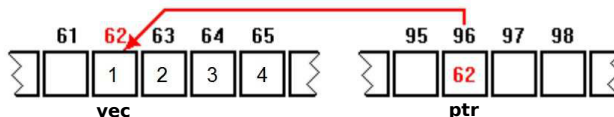
Variable	Address	Memory (byte by byte in hexadecimal)
i	0xffff091c8	bb ee aa ff
b	0xffff091c4	34 12 00 00
c	0xffff091c0	00 00 00 41
x	0xffff091bc	78 56 34 12

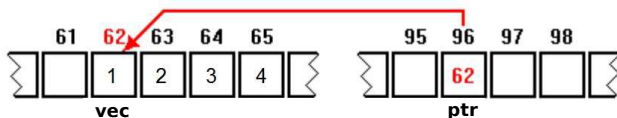
Arrays

```
char vec[4] = {1,2,3,4};  
char *ptr = NULL;
```



```
ptr = vec; /* in the case of arrays, the identifier is the address;  
           so, no need for '&' */
```





- You can do *pointer arithmetic*:

```
ptr++; // ptr now is a pointer to the second element of vec
printf("%d", *ptr); // will print "2"
```

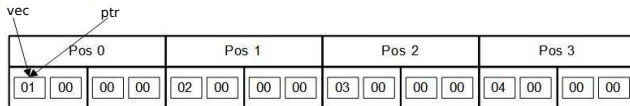
```
ptr+=2; // ptr now is a pointer to the fourth element of vec
printf("%d", *ptr); // will print "4"
```

- The type of the pointer is relevant when performing pointer arithmetic (the compiler decides on the number of bytes added or subtracted based on the pointer type)

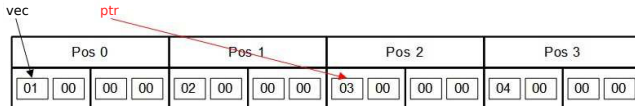
```
int *ptr_total; /*(4 Bytes increments/decrements in IA32)*/  
char *ptr_name; /*(1 Byte increments/decrements in IA32)*/
```

- Example:

```
int vec[4] = {1,2,3,4};  
int *ptr = vec;
```



```
ptr = ptr+2;
```

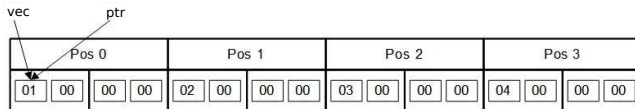


Pointer arithmetic (3/4)

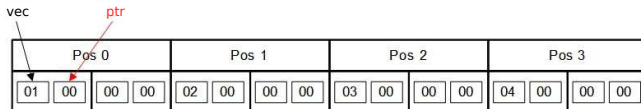
- Now we make `ptr` a `char*`...

```
int vec[4] = {1,2,3,4};
```

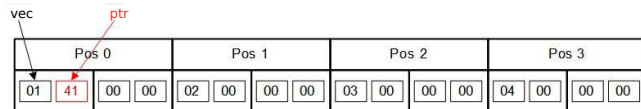
```
char *ptr = (char *)vec; // a cast is needed to avoid a warning...
```



```
ptr = ptr+1;
```



```
*ptr = 'A';
```



- Active learning activity: What is the output of the following code ?
 - A. 1 2 3 4 5
 - B. 2 3 4 5 6
 - C. 1 3 5 7 9
 - D. None of the above.

Listing 2: pointer-arith.c (<http://codepad.org/V3qmKQkp>)

```
#include <stdio.h>

int main(){
    char arr[]={1,2,3,4,5,6,7,8,9,10};
    short *ptr=(short*)arr; // cast is needed to avoid a warning...
    int i;
    for (i=0; i<5; i++) {
        arr[i] = *( ptr + i );
        printf("%d ", arr[i]);
    }
    return EXIT_SUCCESS;
}
```


- Print data to the console using printf and puts

Listing 3: output.c (<http://codepad.org/swgi0sNG>)

```
#include <stdio.h>

int main()
{
    int a = 2, b = 3;
    int *c = &a, *d = &b;
    printf("Value: %d\n", *c); /*prints the value 2*/

    printf("Address: %p (%p)\n", &a, c);
    /* outputs the address of a*/

    if (*c == *d) puts("Same value"); // false
    *c = 3;
    if (*c == *d) puts("Now same value"); // true
    c = d;
    if (c == d) puts ("Now same address"); // true
    return EXIT_SUCCESS;
}
```

- Reading from the console can prove problematic... here are some solutions.

Listing 4: input.c (<http://codepad.org/nmuqfS91>)

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n1=-1, n2=-1, n3=-1;
    int *ptr = &n2;
    char buf[BUFSIZ], c;
    /* read integers using scanf(); pass the address of the var. */
    printf ("Enter a number: ");
    scanf ("%d", &n1);
    printf ("Enter another number: ");
    scanf ("%[^0-9]%d", buf, ptr); // garbage goes to buf...
    /* You will notice that scanf is a bit problematic...
       An option, is to read a string and convert as needed
       First, flush characters scanf() leaves in the input */
    while((c = getchar()) != '\n' && c != EOF) /* discard */;
    printf ("Enter yet another number: ");
    if (fgets(buf, sizeof(buf), stdin) != NULL) {
        n3 = atoi(buf); // returns 0 if conversion fails
        printf ("You entered %d %d %d\n", n1, n2, n3);
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE; /* if reaches here, fgets() failed */
}
```

- ❶ Write the representation in memory, in Big endian and Little endian of the following values:
 - ❶ 0x1188 (16 bits)
 - ❷ 0xff3455b6 (32 bits)
 - ❸ 0x28934def (32 bits)
- ❷ Correct the following code:

Listing 5: exerc01.c

```
int main ()
{
    int * ptr ;
    int i;
    int soma=0;

    for (i=0; i<10; i++)
    {
        scanf("%d", ptr);
        soma = soma + *ptr;
    }
    printf("Soma= %d \n", soma);
    return 0;
}
```