Computer Architecture (Practical Class)

Pointers in C

Luís Nogueira

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

lmn@isep.ipp.pt

2025/2026

Introduction

- A program sees memory as one large array containing bytes
- We do not use the term "index" when refering 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 RV32, **addresses are 32 bits long**, ranging from 0 to $2^{32} - 1$ (4 GiB)
- In RV64, addresses are 64 bits long, ranging from 0 to $2^{64} - 1$ (16 EiB)
    - Most current RV64 implementations provide Sv48 virtual-memory support (48-bit VA), while actual physical-address bits are implementation-defined (often 44 bits, i.e. up to 16 TiB of RAM, or 52 bits, i.e. up to 4 PiB)

- 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 in this class)

Pointers always have the same size

- The size of an address of the underlying architecture (32 bits (4 bytes) in RV32)

- 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

Pointers: Use and notation

- 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:

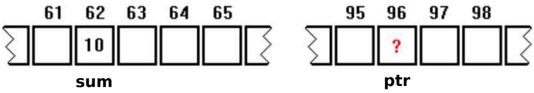
```
*ptr1 = 10;    // assign 10 to the value pointed by prt1
*ptr2 = 'X';   // assign 'X' to the value pointed by prt2
```

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

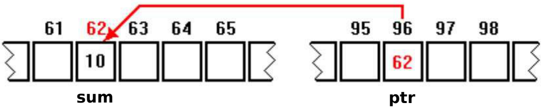
Important note

For the sake of simplicity, in the following schemes, an address is represented in only one byte. DO NOT forget that real addresses **always occuppy 4 bytes in RV32**

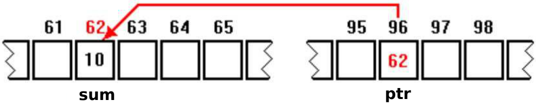
```
char sum;
char *ptr;
sum = 10;
```



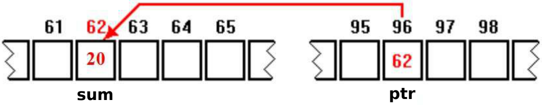

```
ptr = &sum;
```



- 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 way[1]:

```
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 pointed by ptr (x)
```

[1] This 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.

Big/Little Endian

- 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, PowerPC Mac, transferring data on the Internet, ...
- Little Endian
  - Store the least significant byte in the smallest address.
  - Adopted by x86, ARM, RISC-V, ...

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

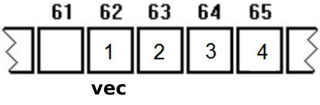
0x12	Address x
0x34	Address $x + 1$
0x56	Address $x + 2$
0x78	Address $x + 3$

Data representation in RV32

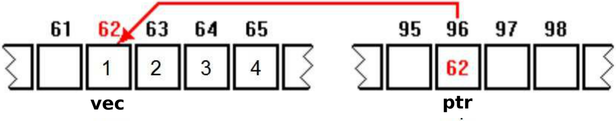
```c
#include<stdio.h>

int main(){
    unsigned int i = 0xFFAAEEBB;
    short 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	0x9854fa7c	bb ee aa ff
b	0x9854fa7a	34 12
c	0x9854fa79	41
x	0x9854fa74	78 56 34 12

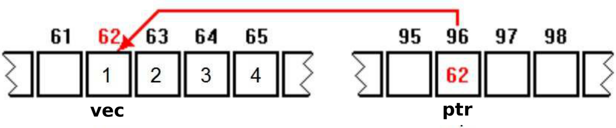
```
char vec[4] = {1,2,3,4};
char *ptr    = NULL;
```




```
ptr = vec;  /* Array names already represent the address of their
               first element, so using '&' is unnecessary */
```



Pointer arithmetic (1/2)



- You can do **pointer arithmetic**:

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

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

Pointer arithmetic (2/2)

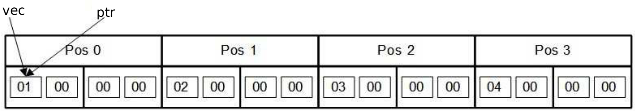
Important note

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)

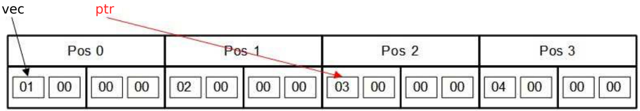
```c
int *total_ptr;         // total_ptr++ moves the pointer by sizeof(int)
                        // (4 bytes)

char *name_ptr;         // name_ptr++ moves the pointer by sizeof(char)
                        // (1 byte)

short *count_ptr;       // count_ptr++ moves the pointer by sizeof(short)
                        // (2 bytes)

long long *big_ptr;     // big_ptr++ moves the pointer by sizeof(long long)
                        // (8 bytes)
```

Pointer arithmetic: Example (1/2)

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

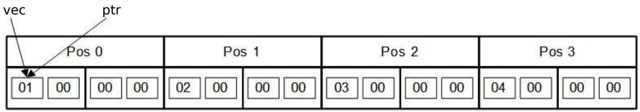



```
ptr = ptr+2;
```

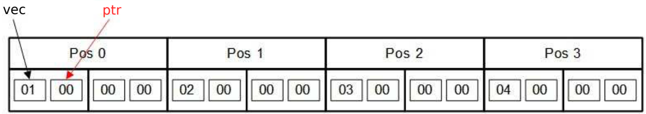


Pointer arithmetic: Example (2/2)

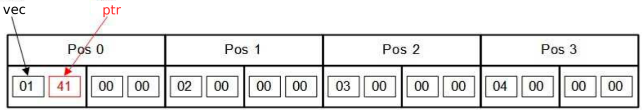
```
int vec[4] = {1,2,3,4};
char *ptr = (char *)vec; // a cast is needed to avoid a warning...
```




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




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



Practice

- 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 1: pointer-arith.c

```
#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("%hhd ", arr[i]);
   }
   return 0;
}
```

Listing 2: output.c

```c
#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);       // prints 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 0;
}
```

Listing 3: input.c

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

int main(){
   int n1 = -1, n2 = -1, n3 = -1;
   int *ptr = &n2;
   char buf[BUFSIZ], c;

   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);
   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
}
```

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

- Correct the following code:

Listing 4: exerc01.c

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

  for(i=0; i<10; i++){
     scanf("%d", ptr);
     sum = sum + *ptr;
  }

  printf("Sum = %d \n", sum);

  return 0;
}
```