Pointers

CSC100 / Introduction to programming in C/C++

Marcus Birkenkrahe

April 29, 2024

README

- This script introduces C pointers in theory and practice.
- This section, including some sample code, is based on: chapter 11 in King (2008), and chapter 7 in Davenport/Vine.

Overview

- Remember the fundamental architectural problem of the "von Neumann" architecture of digital computers.
- Imagine: CPU ≡ brain, Memory ≡ house, Harddisk ≡ moon

    1. CPU and the Brain: The CPU (Central Processing Unit) is like the brain because it is responsible for processing all instructions and making decisions. It performs the core computations and controls other parts of the computer based on the input it receives, similar to how the brain processes sensory information and dictates responses.
    2. RAM and a House: RAM (Random Access Memory) is akin to a house because it serves as the active, working area where tasks are carried out and information is temporarily held. Just as a house contains rooms where daily activities occur (e.g., sleeping, eating, working), RAM holds active applications and data for quick access by the CPU. The size of the RAM affects how much

Figure 1: Roadside stand near Birmingham, AL (LOC@flickr.com)

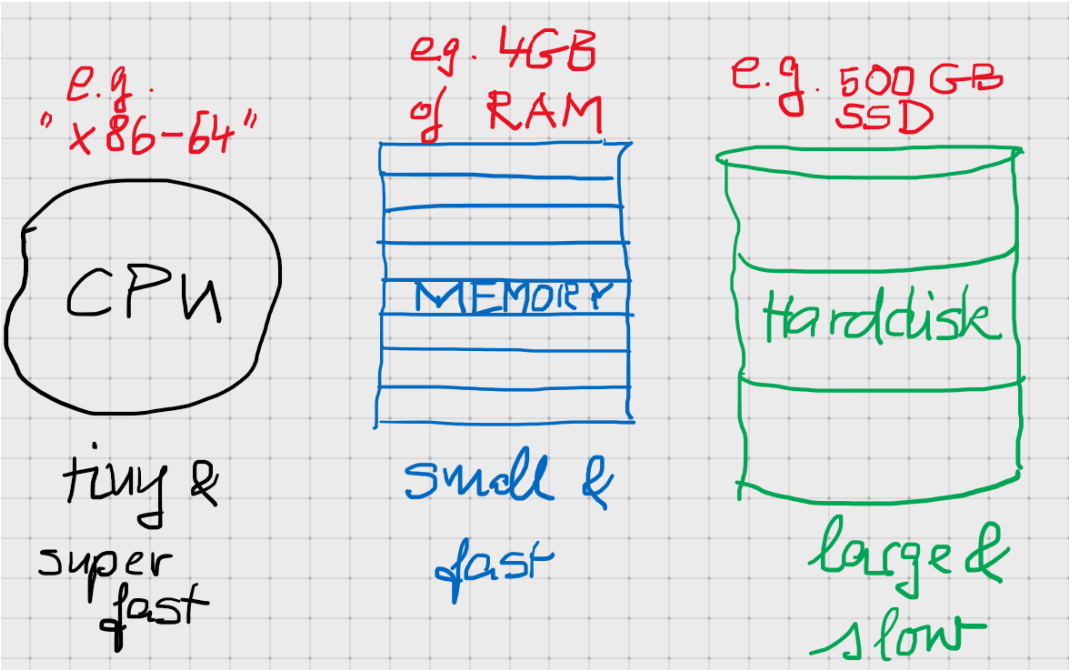


Figure 2: Computer architecture (simplified)

activity (or how many applications) can be handled at once, much like the size of a house determines how many activities can comfortably occur simultaneously.

3. Hard Disk and the Moon: A hard disk is comparable to the Moon in that it serves as a distant but integral storage location. Just as the Moon orbits Earth and is a more remote body holding impacts of meteorites and historical footprints, a hard disk stores data that is not immediately needed for current processing tasks but is crucial for long-term storage. Its contents are not as quickly accessible as those in RAM, reflecting the Moon's greater distance compared to our immediate environment.

- Computer memory is like a list of *locations*
- Each chunk of memory has an *address* to a location
- *Pointers* point to these addresses
- The *address* is not the house, it's a *reference* to the house:
- C#, Java, Python... do not offer pointers (easily) - most of the direct memory management is "abstracted" away (i.e. the memory is hidden and a simpler user interface is offered).

    1. C#: C# does support pointers, but their use is limited to unsafe code blocks. You must explicitly mark these areas with the `unsafe` keyword. This feature allows you to perform operations that involve direct memory access, similar to C or C++. However, using pointers is generally discouraged in favor of safer constructs provided by the language, such as objects and reference types.
    2. Java: Java does not support pointers, at least not in the traditional sense as seen in C or C++. Instead, Java handles memory through references to objects, and these references are abstracted away from direct memory address manipulation. This abstraction is part of Java's design to ensure security and simplicity in memory management, reducing the likelihood of errors such as memory leaks or pointer arithmetic.

Main Memory

Address	Data
0000	Hello
0001	World
0010	This
0011	are
0100	Some
0101	Sample
0110	Data
0111	Stored
1000	Into
1001	An
1010	Imaginary
1011	Main
1100	Memory
1101	Of
1110	A
1111	computer

Figure 3: Sample data stored in an imaginary memory stack

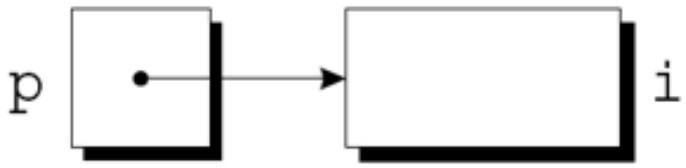


Figure 4: A pointer p points to address of i

Figure 5: envelope = pointer to an address

3. Python: Python also does not use pointers. It handles everything by object references. The language is designed to abstract away most direct memory management tasks from the developer, automatically handling object creation and destruction through a built-in garbage collector. Python's approach simplifies the development process but at the cost of direct control over memory and performance optimizations that pointers can offer.

- C and C++ offer pointer variables and operators naturally
- This gives you a lot more control over the computer (because every operation, every process involves memory management)
- Examples:
  - **String manipulation** (working with text - e.g. when creating fast-performing chat bots or AI agents.
  - **Dynamic memory allocation** - the process of assigning memory during the execution time (when a program typically competes with thousands of other processes).
- This is *mind control*: You can essentially decide what the computer should think with which part of its "brain" (great potential to mess up, too), e.g. when you mis-allocate resources.

Indirection (concept)

- Open Emacs on a codealong file like tinyurl.com/cpp-practice-org and save it (C-x C-s) as an Org-mode file `pointers.org` to code along.
- Imagine you have a *variable* `iResult` that contains the *value* `75`.
- The variable is *located* at a memory address, e.g. `0061FEC8`.

```
int iResult = 75.;
printf("%d is referenced by %p\n", iResult, &iResult);
```

```
75 is referenced by 0x7fffce0fa6c4
```

- You see that repeated executions of this code creates different addresses. A *pointer* is a variable that holds an address.
- Imagine you have a *pointer variable* `myPointer` that contains the address of the variable `iResult`:

```
int iResult = 75.;
printf("%d <- %p\n", iResult, &iResult);
int *myPointer = &iResult;
printf("%p -> %d\n",myPointer, iResult);
```

```
75 <- 0x7ffece47296c
0x7ffece47296c -> 75
```

- This means that `myPointer` *indirectly* points to the value `75`.
- You already worked with addresses: an *array name* `a` is a pointer to the start of the array, the address of `a[0]`.
- In the next code block, the conversion specifier `%p` lets us access the addresses that correspond to elements of the array `a`, and even the address for the whole array.
- We declare an array `a[2]` and then we print its addresses:

```
int a[2] = {100,1000}; // define a 1-dim array of 2 elements
printf("a[0] = %p\na[1] = %p\n&a   = %p\n", &a[0], &a[1], &a);
```

```
a[0] = 0x7fffe36dff40
a[1] = 0x7fffe36dff44
&a   = 0x7fffe36dff40
```

- You can see that the address for `a` points to `a[0]`.
- You already worked with pointers: arguments in the call of `scanf` are *pointers*: without the `&`, the function would be supplied with the *value* of `i`, not the *address*. But `scanf`'s job is to assign a memory location (an address) to the input variable.

```
int i;
scanf("%d", &i);
```

- The relationship between variable value and memory address is called *indirection: A /pointer* provides *indirect* access to the value via the address where the value is stored.

Indirection (code)

- There are two *unary* pointer operators:
  - the *address* (or referencing) operator `&`
  - the *indirection* (or dereferencing) operator `*`
- The unary *address* operator `&` returns a computer memory address, e.g. `&i = 0x7ffc7600b79c` - it *references* the memory location.
- What if `i` has not been initialized yet? Will the address change? Print the reference to `i` before initializing it, and after.

```
int i;
printf("%p\n",&i);
i = 100;
printf("%p\n",&i);

0x7ffd75d787f4
0x7ffd75d787f4
```

- The unary *indirection* operator `*` returns a value, e.g. `*p = 75` if `p` points at a variable that contains the value `75`.
- Let's do it all in one code block:
  1. declare an integer variable `i`
  2. assign the value `1` to `i`
  3. declare an integer pointer `*p`
  4. assign the address of `i` to `p`
  5. print `i` and `&i` (`"is located at address"`)
  6. print `p` and `*p` (`"points to value"`)

```
// variables
int i; // declare integer variable
i = 1; // assign value to variable

// pointers
int *p; // declare integer pointer
p = &i;
```

```
// print variables
printf("%d is located at address %p\n",i,&i);

// print pointers
printf("%p points to value %d\n",p,i);

1 is located at address 0x7ffc17aabbec
0x7ffc17aabbec points to value 1
```

- What if you assign a number `1` to `p` instead of an address?

```
int *p; // declare integer pointer
p = 1; // warning: missing 'cast'

// print pointer value
printf("%p\n",p); // '0x1' is a reserved memory address

0x1
```

- Compiler message (`-Wint-conversion` is a compiler flag):

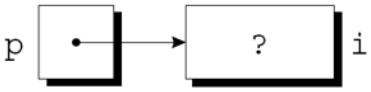
```
warning: assignment to ‘int *’ from ‘int’ makes pointer from integer
         without a cast [-Wint-conversion]
15 | p = 1; // warning: missing 'cast'
   |   ^
```

- Here is more documentation on compiler warnings. You can add them to your code block with the header argument `:flags`, e.g. `:flags -Wall`

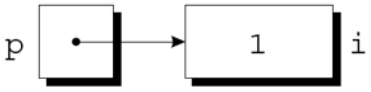
- The figure illustrates these concepts. Can you describe what goes on from line to line?

    1. The pointer `p` points to the address `&i` of the variable `i`.
    2. `i` is initialized with the value `1`. `p` still points at it.
    3. To change the value of `i` indirectly using the pointer `p`, we assign `*p = 2`. The indirection operator `*` designates a pointer.
    4. To check that `i` indeed has been changed, we print it.
    5. `*p` also prints the value of `i`.

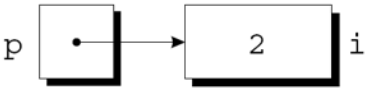
```
p = &i;
```




```
i = 1;
```




```
printf("%d\n", i);     /* prints 1 */
printf("%d\n", *p);    /* prints 1 */
*p = 2;
```




```
printf("%d\n", i);     /* prints 2 */
printf("%d\n", *p);    /* prints 2 */
```

Figure 6: Graphical illustration of the indirection operator (Source: King)

* and & are inverse to one another

- Address and indirection operator are *inverse* to one another (i.e. they reverse each other's operation - applying both amounts to doing nothing).
- Applying indirection `*` to an address *dereferences* it.
- Applying referencing `&` to a pointer extracts its address.

```
// declaring and initializing
int val = 75, *ptr = &val;

// print variable and dereferenced pointer
printf("value = %d => *&(value) = %d\n",
        val, *&val);

// print pointer and address of pointer
printf("pointer = %p => &*(ptr) = %p\n",
       ptr, &*ptr);
```

```
value = 75 => *&(value) = 75
pointer = 0x7fff765d52ac => &*(ptr) = 0x7fff765d52ac
```

- Applying `*` to the pointer takes us back to the original variable (dereferences the pointer)

```
j = *&i  // same as j = i
```

Pointers must be initialized

- Non-initialized pointers lead to invalid data or expressions.
- Pointer variables should always be initialized with:
  - another variable's memory address (e.g. `&i`), OR
  - with 0, OR
  - with the keyword `NULL`.
- Here are some *valid* pointer initializations - `printf` uses the conversion specifier `%p` for pointers.

```
double *ptr1;  // pointer declarations
int *ptr2;
int *ptr3;
double x = 3.14; // initialize variable

ptr1 = &x; // initialize with address
ptr2 = 0;  // initialize with 0
ptr3 = NULL; // initialize with NULL

printf("%p %p %p\n", ptr1, ptr2, ptr3);

0x7ffc6b2ea878 (nil) (nil)
```

- Let's print these last values: how do you have to change the `printf` statement? (Add the flag `-w` to disable all warnings)

```
double *ptr1;  // pointer declarations
int *ptr2;
int *ptr3;
```

```
double x = 3.14; // initialize variable

ptr1 = &x; // initialize with address
ptr2 = 0;  // initialize with 0
ptr3 = NULL; // initialize with NULL

// different conversion specifiers
printf("%.2f %d %d\n", *ptr1, ptr2, ptr3);

3.14 0 0
```

- Here are a few non-valid initializations: we want to change the value of a variable using the pointer to its memory address.
  - can you tell why?
  - can you right the wrong?
  - print `iPtr`, `&i` and `i`

```
int i=5; // declare integer i
int *iPtr; // declare pointer iPtr

iPtr = &i; // initialize pointer
iPtr = 7;  // change value of variable
```

- Solution:

```
int i;
int *iPtr;

iPtr = &i;   // pointer initialized with memory address
*iPtr = 7;   // value of i indirectly changed

printf("%p %p %d\n", iPtr, &i, i);

0x7ffd5bcfacfc 0x7ffd5bcfacfc 7
```

Let's practice!

- Download and complete the practice file: tinyurl.com/cpp-pointers-practice

References

- Davenport/Vine (2015) C Programming for the Absolute Beginner (3ed). Cengage Learning.
- Kernighan/Ritchie (1978). The C Programming Language (1st). Prentice Hall.
- King (2008). C Programming - A modern approach (2e). W A Norton. URL: knking.com.
- Orgmode.org (n.d.). 16 Working with Source Code [website]. URL: orgmode.org