C Lab 5
Pointers and Memory Allocation - 1

COMP 310 / ECSE 427 : Fall 2024

Jason Zixu Zhou

Revised from the slides of Junyoung/“Clare” Jang

Introduction to Pointers

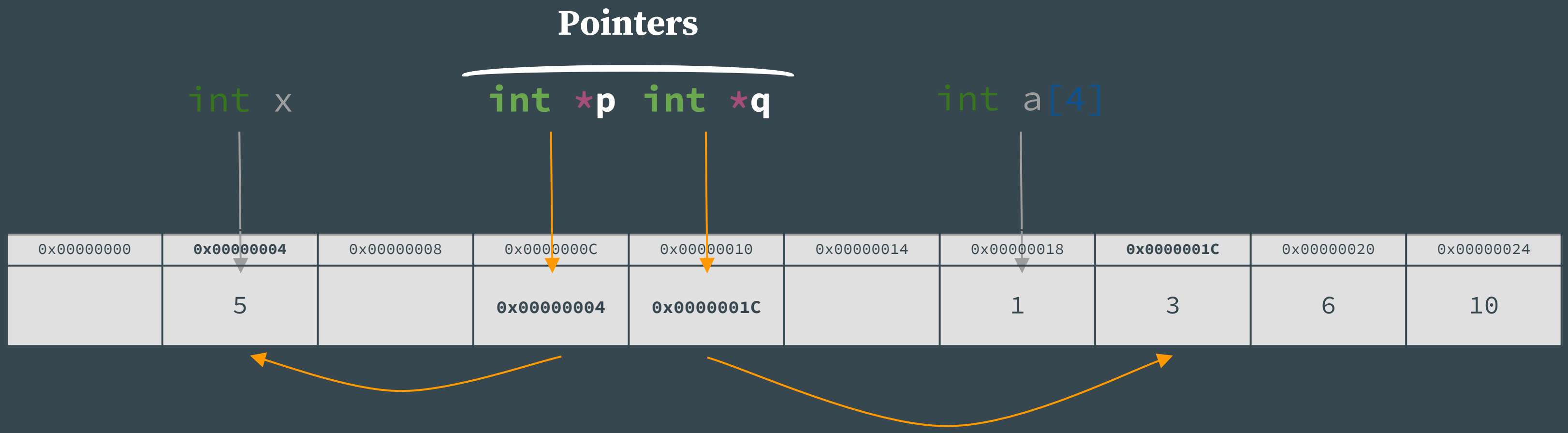
Pointer

Pointer : a **variable** that stores an **address** in memory

Address in memory?
For ex)

- The location where a variable stores its content
- The location of the 2nd item of an array
- The location where a function stores its code

Pointer in Picture



Definition of a Pointer Variable - Simpler Version

`<type> *<identifier>;`
or `<type> *<identifier> = <initial value>;`
For ex)

- `int *p`
- `int *q = p`
- (multiple-variable definition) `int *a, *b = q, *c`

Here, `<type>` is the type of value in the stored address

Getting an Address

1. `&` *operation* : getting an address of a variable, array item, or function
   For ex) `&x`, `&a[2]`, `&printf`
2. Using array; the value of array is already an address!
   For ex) `int a[3]; …; int *p = a;`
3. `+` *operation* : translating an address by a given offset
   For ex) `p + 2`, `&a[0] + 1`
4. `NULL` *macro* : giving a special "null-pointer"
5. #define PI 3.14 creates a constant PI
6. *library functions for memory allocation*

Having some danger of UB!!
(Undefined Behaviours)

Printing an Address

`%p` *specifier* : a specifier for printing an address value

For ex) `printf(“The address of variable x is: %p\n”, &x);`

cast the pointer to (void *) when using %p to ensure portability and correctness in different environments.

Exercise 1

Suppose that we have the following code

```
int a[4] = { 4, 3, 2, 1 };
int *p = a;
printf(“p is %p\n”, p);
printf(“p + 6 is %p\n”, p + 6);
```

1. What is the difference between printed addresses? Why do they have that difference? (Note that they are usually in hexadecimal format)
2. In this code, `p + 6` is actually a dangerous operation (UB). Can you guess why?

Using an Address

You should not use this with the null-pointer in most cases!

1. `*` *operation* : Dereferencing to get a value

   For ex) `printf("%d\n", 5 + *p);`

2. `*` *operation* : Dereferencing to assign a value

   For ex) `*p = 3 * 7;`

3. `[n]` *operation* : a syntactic sugar for `*` and `+`

   For ex) `p[i]` is a shorthand for `*(p + i)`

Needs for Pointers - 1

C functions follow “call-by-value” & “return-by-value”; when a function is called:

1. Evaluate all the arguments
2. Copy the values of arguments to the parameter variables
3. Compute the body of the function
4. Copy the return value and return it

Under this strategy,
how can we implement a function that modifies a variable outside its body?

Needs for Pointers - 2

For example, suppose that we want to write the `swap` function that satisfies:

- After `swap(x, y)` is called with two integer variables `x` and `y`, `x` has the value of the previous `y` and `y` has the value of the previous `x`

```c
void swap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

Needs for Pointers - 3

C-implementable specification for `swap` function is:

- After `swap(&x, &y)` is called with two integer variables `x` and `y`, `x` has the value of the previous `y` and `y` has the value of the previous `x`

```
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

Exercise 2

Let’s re-implement the standard library function `strcpy`

1. After `myStrcpy(dst, src)` is called, `dst` should have the same string as `src`
2. The function should return `dst`
3. Its header is: `char *myStrcpy(char *dst, char *src)`
4. Recall that C strings always terminate with `‘\0’`

Memory Allocation

Static Allocation of Memory

When we allocate memory statically, we know the size of memory.
For ex)

- `int a;` — we need 4 bytes in x86-64 Linux
- `char b[24];` — we need 24 bytes in x86-64 Linux
- `int a; int b;` — we need 8 bytes in x86-64 Linux

Needs of Dynamically Allocated Memory

When we make an OS, is it possible to pre-calculate the size of the memory for all processes?

When we make a student management solution for any classes, is it possible to pre-calculate the size of the memory for all student entries?

Dynamic Management of Memory - 1

We can use standard library functions from `<stdlib.h>` to manage memory dynamically

- `malloc` : **allocate** a memory
  For ex) `malloc(10 * 4)` — allocate 40 bytes, i.e. 10 items of 4 bytes

- `calloc` : **allocate** a memory and **initialize** its content with `0`s
  For ex) `calloc(10, 4)` — allocate 10 items of 4 bytes whose contents are `0`

- `realloc` : **allocate** a **possibly-new** memory of a size while keeping content
  For ex) `realloc(p, 3 * 4)` — get a pointer to 12 bytes which contains the same content as `p` up to 12 bytes

Dynamic Management of Memory - 2

As a compiler cannot guess how long we will use an allocated memory, we should manually de-allocate it as well.

- `free` : **de-allocate** a memory allocated by one of the previous functions
  For ex) `free(p)` — de-allocate `p`. the value of `p` should be an address to a memory allocated by one of the previous functions.

Dynamic Management of Memory - 3

What happens if we allocate a memory and then never free it?

The memory will remain “in-use” until the process will terminate. This issue is called “memory leak”

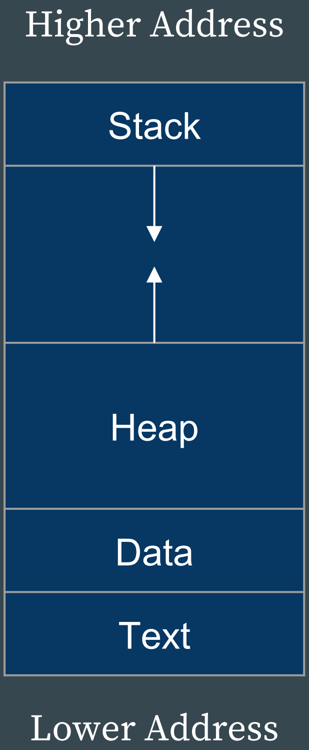
If memory leak is accumulated and the process is a long-run process like a web server, it can cause Out-Of-Memory (OOM).

Dynamic Management of Memory - 4

Note:

- Statically allocated memory fragments: live **outside the heap**
- Dynamically allocated memory fragments: live **in the heap**

This makes static allocation has a quite restrictive size limit. Thus, even when one wants to use one big memory that persists over the lifetime of a process, they need to use dynamic allocation.



Exercise 3

Let’s implement a program that reads

1. the number of courses
2. the score of each course in the range of [0,100]

from the user input, and then prints all scores and their average. Note that the program should work with a huge number of courses as well.

Don’t forget to `free` the allocated memory!