C Pointer Tutorial

Resources

- The C Programming Language: The bible of C. Read the Chapter 5 if there is anything about pointer confused you.
- https://cdecl.org/: The website to go for explanations of pointers.
- https://github.com/root-project/cling: I recommend to use Cling to run and play with following codes.
- Campuswire and Office Hours: If you don't know the answer of practice questions, ask them in campuswire or office hour. We are very glad to explain more details to you.

Fundamentals

the pointer

Definition: a pointer is a variable that contains the address of a variable.

```
// important to include stdio.h in Cling as well, otherwise pris
#include <stdio.h>
// num --> name, 42 --> value, 0x7fff5694dc58 --> address
int num = 42;
// int* to declare a memory pointer of int, & operator to get the address of an variable
// print a pointer with %p
int *addr = #
printf("%p\n", addr);
// Output: 0x7fff5694dc58
// print the value of pointer
printf("Value of pointer: %d, Actual value: %d\n", *addr, num);
// Output: Value of pointer: 42, Actual value: 42
*addr = 1234;
printf("Value of pointer: %d, Actual value: %d\n", *addr, num);
// Output: Value of pointer: 1234, Actual value: 1234
In the code above, we are dealing with operators of pointers: - int *, char
*, double *, void * -> declares pointer of certain type - size of pointers are
same because they are memory addresses - For 32bit systems sizeof(int *) =
sizeof(char *) = sizeof(void *) = 4 bytes - For 64bit systems sizeof(int
*) = sizeof(char *) = sizeof(void *) = 8 bytes - *pointer -> derefer-
ence the pointer to get value - * operator is overloaded in definition, just to
remember the following rule to defferentiate: - If * is used with type, it's a
decleration of pointer of the type - If * is used with a variable name, it is the
deference of the pointer value - & \rightarrow get the address of a variable - %p \rightarrow print
```

Practice

- As for pointer declaration, both int* foo and int *foo are valid for compilers. Research on the style of C pointer, and pick your way of writing.
 - Great discussions here: https://stackoverflow.com/questions/398395/whyis-the-asterisk-before-the-variable-name-rather-than-after-the-type.
- 2. Write a function that implement division and returns both quotient and remainder. You have to use the following function prototype int divide(int divident, int divisor, int* remainder) or void divide(int divident, int divisor, int* remainder, int* quotient).
- 3. Explore the lifetime of local variables when you use pointers. Reason why the following function is wrong.

```
int *hello() {
    int a = 42;
    return &a;
}
int main(int argc, char const *argv[])
{
    int *result = hello();
    // do other things
    printf("%d\n", *result);
    return 0;
}
```

Array and String

Definition: arrays are values stored in a continuous sequence of memory. Definition: string is an array of **char**.

```
int nums[] = {1, 2, 3};
int empty[5];
char str[] = "hello";
char name[] = {'h', 'e', 'l', 'l', 'o'};
printf("%p, %p, %p, %p, %p, %p", nums, &nums, &nums[0], &nums[1], &nums[2]);
// Output: Ox111acceb8, Ox111acceb8, Ox111acceb8, Ox111accebc, Ox111accec0, Ox111accec4, Ox.
// nums = &nums = &nums[0], and each following element increment by 4 (which is sizeof(int),
printf("%d, %d, %d, %d, %d", *nums, *(nums + 1), *(nums + 2), *(nums + 3), *(nums + 4));
// Output: 1, 2, 3, 1221167428, 28
// array reference is essentially pointer arithmetic, so nums[1] = *(nums + 1)
// All the data beyond nums[3] are garbage data, since they are not initialized. They can be
```

```
// A helper loop to print out everything in an array
for (int i = 0; i < sizeof(nums) / sizeof(int); i++) {
    printf("%d\n", nums[i]);
}</pre>
```

Practice

- Print out value of 1[nums] and explain why it is working. In addition, try more combinations like 0[nums] and (-1)[nums + 1].
 - Hint: you can investigate the case by printing the address &(1[nums]).
- 2. Print out size of the two char array above with sizeof(str) and sizeof(name). Explain why their sizes are different.
 - Hint: try to print out all values in the two array.
- 3. Explore the empty array for index from 0 to 20.
- 4. Research on **sizeof** operator (why we don't call it a function?), and explain why how it's working.
 - Hint: run the following code to see how len_arr function cannot give the correct result.

```
size_t len_arr(int *arr) {
    return sizeof(arr) / sizeof(int);
}
int nums[5];
printf("%zu\n", len_arr(nums));
printf("%zu\n", sizeof(nums) / sizeof(int));
```

Address Arithmetic

```
int nums[10];
int *addr = nums;
char name[10];
char *pp = name;
int other[10];
```

```
// Address ± number --> notice here that int pointer shift by 4 bytes and char pointer only
printf("%p, %p, %p\n", addr, addr + 1, addr - 1);
// Output: Ox108b1f170, Ox108b1f174, Ox108b1f16c
printf("%p, %p, %p\n", pp, pp + 1, pp - 1);
// Output: Ox108b1f230, Ox108b1f231, Ox108b1f22f
```

```
// Address ± Address --> notice here we printf by %ld rather than %p
printf("%ld\n", other - addr)
// Output: 88
```

We declare pointers as different types, but as a static weak typed programming lanauge, C allows program to cast pointer types. The following example from The C Programming Language shows us why we need to cast types of pointers.

memcpy is a function in string.h library that allows codes to copy "numBytes" bytes from address "from" to address "to". We are going to implement the function here. The function casts pointers to char * so that we can copy the memory 1 byte at a time (remember sizeof(char) = 1).

```
void memcpy(void *to, const void *from, size_t numBytes) {
    char *cto = (char *)to;
    char *cfrom = (char *)from;
    for (int i=0; i < numBytes; i++)
        cto[i] = cfrom[i];
}
int a[5] = {1, 2, 3, 4, 5};
int b[100];
memcpy(b, a, sizeof(a));
printf("%d, %d, %d, %d\n", b[0], b[1], b[2], b[3], b[4]);</pre>
```

Practice

- 1. Research on the difference between nums and &nums of an array. In particular, explain why nums + 1 and &nums + 1 are different.
 - Hint: do the experiment on int* a[3] and int* b[10].
- 2. Write your own version of memcmp with the following function prototype int memcmp(const void *s1, const void *s2, size_t n).

Function Pointer

Function can be viewed as a procedure with typed inputs and outputs. In C, we define pointers to function based on types of function's arguments and return values.

```
int multiply(int a, int b) {
    return a * b;
}
// Syntax of a function pointer --> return_type (*pointer_name)(argument_type...)
int (*func_pointer)(int, int) = multiply;
printf("%d\n", func_pointer(3, 5));
// Output: 15
```

With the basic introduction of the syntax and usage of function pointer, let's get a deep dive into function pointers with an example.

```
// Follow the codes above, let's accepts function as arguments
int two_numbers(int a, int b, int (*func)(int, int)) {
    return func(a, b);
```

```
}
printf("%d\n", two_numbers(6, 7, func_pointer));
// Output: 42
printf("%d\n", two_numbers(6, 7, multiply));
// Output: 42
// Then like in modern language, we can write function factory that returns a function
int (*func_factory(int a))(int, int) {
    printf("With parameter %d", a);
    int (*func)(int,int) = multiply;
    return func;
}
// To make it more clear, we should use typedef
typedef int (*my_func)(int, int);
my_func func_factory(int a) {
    printf("With parameter %d", a);
    int (*func)(int,int) = multiply;
    return func;
}
  1. Follow the steps of -The C Programming Language- 5.11 Pointers to
     Functions to implement a generic quick sort.
```

- 2. Bonus: research the partial function support with GCC to enable currying and higher order functions.
 - http://gcc.gnu.org/onlinedocs/gcc/Nested-Functions.html