CS1010 Laboratory 05

Pointers, Call-Stack, Arrays, Exercise 3

Zhang Puyu

Group BD04

October 3, 2024

Plan of the Day

1 Pointers

2 Arrays

3 Selected Problems from Exercise 3

Pointers

- Pointers are **variables** just like integers or floating points.
- The "value" stored in a pointer is the **memory address of another variable**.
- Pointer operators:
  1. `&` (address operator): `&x` will **get the memory address** of the variable `x`.
  2. `*` (indirection operator): `*ptr` will **get the value of the variable at address `ptr`**.
  3. `+` and `-` (pointer arithmetic): `ptr + n` and `ptr - n` get the **memory address at *n* positions after/before** `ptr`.
- **Question**: Does a pointer has a memory address? **Yes** because **a pointer is a variable!**.

Interpretation of Pointer Variables

We can interpret the declaration `long *a` in two ways:

1. Treating `long*` as a data type, so "`a` is **the memory address of a `long` varaible**"
2. Treating `*a` as a whole, so "`a` **when dereferenced is a `long` variable**" (and thus it **points to a `long`**).

Since a pointer is a variable, it also has its own memory address! So we can have a **pointer to a pointer** `long **b` (this can go infinitely).

Challenging Question: How to interpret `long **a[3]`?

Array Decay

Consider the following fixed-sized array:

`long arr[5]` $:=$

0	1	2	3	4
1	1	2	3	5

- In C, an array is "anchored" to the **memory address of its 1st element**. In this example, `arr == &arr[0]`.
- Reason why `arr` and `&arr[0]` are equivalent: a fixed-sized array is stored in a **contiguous region** in the computer's memory (meaning there's no "gaps" between any two elements in the memory).
- So what the notation `arr[k]` actually means is: **retrieve the value stored** *k* **locations after the address of** `arr[0]`.
- **Question**: If we try to retrieve `arr[-1]` or `arr[5]`, will it leads to errors in the program? **Not necessarily** because an array in C does not have a well-defined "end point"!

Array-out-of-Bound Error

Consider the following array A:

$$\{\ 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9\ \}$$

Question: What will happen if we try to retrieve `A[10]`? How about `A[-1]`?

(A) Nothing will happen.

(B) The program crashes.

(C) Gibberish values will be retrieved.

(D) Cannot determine for sure.

Correct answer: D :O

A has a **fixed size** of 10 after declaration, so its index ranges from 0 to 9 only. Therefore, both `A[10]` and `A[-1]` would try to access something which is **outside of the array**!

The above is known as an **Array-out-of-Bound** error.

`cp -r ~cs1010/lab05 .`

The directory `lab05` contains an example program `oob.c` to illustrate the Array-out-of-Bound error.

First, compile the files using

```
CFLAGS=@no_sanitize.txt make oob
```

(This *suppresses* several warnings which would have been triggered when using the CS1010 compiler.)

oob.c

```c
#include "cs1010.h"

void foo(long n) {
  long x[3];
  x[n] = 1;
}

int main() {
  long n = cs1010_read_long();
  foo(n);
  cs1010_println_long(n);
}
```

Run `oob` (using `./oob`) with the following inputs respectively:

$-1, 0, 1, 2, 5, 7.$

What do you observe?

Some Observations

- $0, 1, 2$: Correct outputs.
- -1: No error.
  - `A[-1] == *(A - 1) == *(&A[0] - 1)`.
  - There might be some random value stored at this memory address.
  - So accessing this memory address is legal.
- 5: `Segmentation fault (core dumped)`.
  - One of the “most mysterious” errors in C programs.
  - Occurs when the program accesses a memory address with restricted accessibility.
- 7: Incorrect output.
  - `A[7] == *(A + 7) == *(&A[0] + 7)`.
  - However, it happens that `n` is stored at the memory address `&A[0] + 7`.
  - So by changing `A[7]`, we actually change `n`.

Conclusion: manipulating out-of-bound elements of an array is **dangerous**!

Address Sanitizer

Remove `oob` (using `rm oob`) and re-compile using `CFLAGS=@sanitize.txt make oob`

Now,
running `oob` with -1 causes the program to crash (as intended).

Runing `oob` with 3 produces a verbose crash message (but here's a way to read and interpret it).

Why Array Decay Is Useful

Consider the following function:

```
void foo(long n) {
    n += 1;
}

int main() {
    long n = 10;
    foo(n);
    cs1010_println_long(n);
}
```

What will be printed? **10, NOT 11**.

- The `n` in `main` and the `n` in `foo` are actually **different** variables although they have the same name!
- The program will actually create a **local copy** of the value of `n` in the local stack frame of `foo`.
- This is known as **pass by value**.
- **Variable `n`** is not the same as **value of `n`**. We only use the value of `n` without doing anything to the variable itself.

Why Array Decay Is Useful

However, for arrays:

```
void foo(long a[10]) {
    a[0] += 1;
}

int main() {
    long a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    foo(a);
    cs1010_println_long(a[0]);
}
```

1 will be printed.

- Creating a local copy of an array is expensive because the array may have a large size.
- So we always pass in an array as a **pointer**. This is known as **pass by reference**.
- We tell `foo` the location where it can find `a` so that it can do whatever it wants onto `a` directly!
- Anything done to the elements in `a` by `foo` will be reflected in `main`.
- A common usage of this feature: **use array to output multiple values from one single function!**

counter.c

- We want to record the **number of occurrences** of each integer between 0 and 9 in the digits of *n*.
- Essentially, we wish to have a **frequency table**:

digit	0	1	2	3	4	5	6	7	8	9
frequency										

- But isn't this just an array?
- So we just need to increment `freq[n % 10]` by 1 and update `n` by `n /= 10` repeatedly until *n* becomes 0.

`largest.c` (Revisited)

- Sub-problem: given an array of *k* single-digit integers, can we **join** them together to form a *k*-digit integer?
- It suffices to **sort** the digits of *n* in non-ascending order.
- Easy but slower way:
  1. Iterate the array from left to right.
  2. For each index `i`, we try to swap `A[i]` to the left until the left neighbour is greater than or equal to it or there is no more left neighbour.
  3. How to swap `A[i]` with `A[j]`?
     - `A[i] = A[j]` followed by `A[j] = A[i]` — **Wrong**.
     - Need a temporary variable to store the value of `A[i]`.
  4. This is known as insertion sort.
  5. Note that in the **worst case**, every element `A[i]` needs to be swapped *i* times.
  6. In total this leads to $\frac{k(k-1)}{2}$ swaps for *k* integers.

largest.c (Revisited)

- Sub-problem: given an array of k single-digit integers, can we **join** them together to form a k-digit integer?
- It suffices to **sort** the digits of n in non-ascending order.
- (A lot) faster way:
  1. Iterate the array from left to right.
  2. Record the **frequencies** of integers 0 to 9 into the following table:

digit	0	1	2	3	4	5	6	7	8	9
frequency										

  3. From 9 down to 0, append $f(i)$ copies of integer i to the end of the output integer.
  4. This is known as counting sort.
  5. Note that for k integers, it takes k steps to construct the frequency table and another k steps to join the integers back.
- But how can negative inputs be addressed?
- Notice that if $n < 0$, then $n_{\max} = -(-n)_{\min}$.

days.c

- Consider the following table:

1	2	3	4	5	6	7	8	9	10	11	12
31	28	31	30	31	30	31	31	30	31	30	31

- Notice that there's an offset of 1 on the array indices.
- The day of the year of the d-th day of the m-th month is just:

$$\text{total number of days in the first } (m-1) \text{ months} + d.$$

`ID.c`

- A **character** is enclosed with **single quotation marks**, e.g., `'A'`.
- Like other types, you can create an array of characters using `char c_array[10]`.