CS1010 Laboratory 06

Pointers, Strings, Heap, Exercise 4

Zhang Puyu

Group BD04

October 10, 2024

Plan of the Day

Before Everything Else...

Consider

```
if (n > 0) {
    // do something
    return n;
}
// other stuff
if (n <= 0){
    // do something else
    return -n;
}
```

Is this correct?

It seems that either $n > 0$ or $n \leq 0$. But it is in general **a bad practice** to not have a default return statement outside of all conditional branches.

For example, between the two conditional blocks, if we modify the value of n from negative to positive, then both conditional blocks will be bypassed!

counter.c

Baseline implementation:

```
if (n == 0) {
    freq[0] += 1;
} else {
    while (n != 0) {
        freq[n % 10] += 1;
        n /= 10;
    }
}
```

But we can remove the `if` block by using

```
do {
    freq[n % 10] += 1;
    n /= 10;
} while (n != 0);
```

Linear VS Divide & Conquer

Consider these 2 functions to find the maximum among an array of n elements:

```
long find_max_1(long* list, long start, long end) {
    if (start == end - 1) {
        return list[start];
    }
    long local_max = find_max(list, start, end - 1);
    return local_max > list[end - 1] ? local_max : list[end - 1]
}
long find_max_2(long* list, long start, long end) {
    if (start == end - 1) {
        return list[start];
    }
    long mid = (start + end) / 2;
    long left_max = find_max(list, start, mid);
    long right_max = find_max(list, mid + 1, end);
    return left_max > right_max ? left_max : right_max;
}
```

In `main`, we call both functions with `(list, 0, n)`. **How do the 2 algorithms behave differently**?

Linear VS Divide & Conquer: Analysis


```
long find_max_1(long* list, long start, long end) {
    if (start == end - 1) {
        return list[start];
    }
    long local_max = find_max(list, start, end - 1);
    return local_max > list[end - 1] ? local_max : list[end - 1]
}
```

Here we are saying: in order to determine whether the i-th number is **the biggest among the first i terms**, we will first find **the maximum among the first $(i-1)$ terms**.

Note that this is the same as **iterating the whole array from index 0 to index $n-1$!**

So this algorithm is more prone to **stack overflow**.

Linear VS Divide & Conquer: Analysis


```
long find_max_2(long* list, long start, long end) {
    if (start == end - 1) {
        return list[start];
    }
    long mid = (start + end) / 2;
    long left_max = find_max(list, start, mid);
    long right_max = find_max(list, mid + 1, end);
    return left_max > right_max ? left_max : right_max;
}
```

Here, we say that: to find the maximum among the n elements, we first find **the maximum among the first $\frac{n}{2}$ elements**, and then find **the maximum among the last $\frac{n}{2}$**, outputing the bigger of the two.

Try to convince yourself: it takes at most $\log_2 n$ **divisions** to reduce an array of size n to a singleton. (So at any point of time at most $\log_2 n$ function calls are in the stack.)

Question: **Is divide and conquer necessarily faster**?

An Optional Question

Let's consider a small variation of the problem:

Instead of searching for the maximum/minimum, I now wish to determine whether a particular element v exists in an array of size n. How will the two algorithms perform?

What if the input array is **sorted** in ascending order? Which one of the two algorithms is faster?

padovan.c

The standard approach is as follows:

- Compute from $p(0)$ to $p(n-100)$ using the **sliding window** approach similar to `fibonacci.c`, but **do not store the values in an array yet!**
- From $p(n-99)$ onwards, store the computed value in an array.
- Print the array in descending order of index.

But is there a more elegant approach?

`padovan.c`: Circular Arrays

Suppose we have a **circular list** of size 100, i.e. the 101-th element will overwrite the first element and so on.

Then we do not need to care from which index onwards to store in the output list!

We can then just store the Padovan Numbers in sequential order, and after all n numbers have been stored, the list is guaranteed to contain the last 100 numbers!

How to implement this: store the i-th Padovan Number (0-based index) into `list[i % 100]`!

Unfortunately, this cannot pass the running time requirement because the modulo operation is a bit slower than the other arithmetic operations.

Circular Array Pseudocode


```
Initialise a = [0, 1, 1, 0, 0, ..., 0]
where size(a) = 100

for i = 3, 4, 5, ..., n - 1, do:
    prev = a[(i - 2) % 100]
    curr = a[(i - 1) % 100]
    next = padovan(prev, curr)
    a[i % 100] = next

for j = 1, 2, ..., 100, do:
    print a[(n - j) % 100]
```

cp -r ~cs1010/lab06 .

We will use these to demonstrate heap-related bugs.

`nobug.c` contains the bug-free version of `bug1.c` to `bug4.c`.

```c
#include "cs1010.h"

int main() {
  size_t n = cs1010_read_size_t();
  long *ar = cs1010_read_long_array(n);
  if (ar == NULL) {
    cs1010_println_string("memory allocation failed");
    return 1;
  }
  for (size_t i = 0; i < n; i += 1) {
    cs1010_print_long(ar[i]);
  }
  free(ar);
}
```

Heap Buffer Overflow (`bug1.c`)

This happens when we try to access indices which have not been allocated, i.e. the counterpart of array-index-out-of-bound in dynamic arrays.

When compiled without `address sanitizer`: an extra number was output.

Memory Leaks (`bug2.c`)

This happens when we allocate something onto the heap but forget to free the memory before our program exits.

When compiled without `address sanitizer`: No issue. (**So why is this even relevant?**)

Real-life example: poorly designed software making our computers laggy. One major possible reason is that they drain our available memory by not freeing it to the heap.

Different programming languages deal with this differently, for example:

- **Java** uses a garbage collector.
- **Python** uses reference counting.

Double Free (`bug3.c`)

This happens when we free the same pointer more than once, i.e., we are freeing some memory which has already been freed previously.

When compiled without `address sanitizer`: the program crashes.

Heap Use after Free (`bug4.c`)

This happens when we try to use a pointer after the memory allocated to it has been freed.

When compiled without `address sanitizer`: arbitrary output.

Attempting Free on Address Which Was Not `malloc()`-ed (`bug5.c`)

This happens when we free a pointer which is allocated on the stack, not the heap.

When compiled without `address sanitizer`: the very unhelpful `Segmentation fault (core dumped)`.

Stack Buffer Overflow (`bug6.c`)

This happens usually when we try to print a string which is stored in an array with a size smaller than the string's length.

When compiled without `address sanitizer`: random characters printed.

Strings

- Strings in C use **double quotation marks** while single quotation marks are for characters. E.g. `"A"` and `'A'` are different!.
- Every string ends with the **null** character `'\0'` which is invisible. This is to indicate **at which memory address the string terminates**.
- **Question**: Why is it necessary to use `'0'`? (So troublesome!)
- **Question2**: What is the difference between `char *str = "A"` and `char str[2] = "A"`.

up.c

- `size_t`: **unsigned** `long`, i.e. **positive** `long` integers. (Side question: do you think that the maximum number allowed to be stored as `long` is the same as `size_t`?)
- The idea is simple: iterate the string; for each character check whether it is lower-case; if it is lower-case, capitalise it. But the questions are
  - How do we know **when to stop** scanning for characters?
  - How do we check **whether a character is lower-case**?
  - How do we **capitalise** a lower-case character?

up.c

- Every string ends with the **null character**, so if the current character is `'\0'`, we know that the string has been fully iterated.
- Every character is essentially an **integer** (i.e., its **ASCII** value), so we can simply use `'a' <= c <= 'z'` to determine whether a character `c` is a lower-case letter.
- In the ASCII table, upper-case letters are placed right after lower-case ones in sequential order, so we can just offset a lower-case letter by `'A' - 'a'` (which is 26) to capitalise it.

length.c

- Note that you only need to edit `length.c`, the `main` function is located at `length-main.c` which you are **not allowed** to change!.
- This is just to count the number of elements in a character array.
- Do we include the null character as part of the length?

concat.c

- Given two arrays, how do we concatenate them into a new bigger array?
- Any difference when we are concatenating two strings?
- Is the resultant string a fixed-size array or a dynamic array?
- Note that if the lengths of the two strings are m and n respectively, the concatenated string will have a length of $m + n - 1$, not $m + n$! (Why?)
- Don't forget the final null character.