CS 61C Summer 2023

C

Discussion 2

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 True or False: C is a pass-by-value language.

1.2 The following is correct C syntax:

```
int num = 43
```

1.3 In compiled languages, the compile time is generally pretty fast, however the run-time is significantly slower than interpreted languages.

1.4 The correct way of declaring a character array is `char[]` array.

1.5 Bitwise and logical operations result in the same behaviour for given bitstrings.

1.6 What is a pointer? What does it have in common to an array variable?

1.7 If you try to dereference a variable that is not a pointer, what will happen? What about when you free one?

1.8 Memory sectors are defined by the hardware, and cannot be altered.

1.9 For large recursive functions, you should store your data on the heap over the stack.

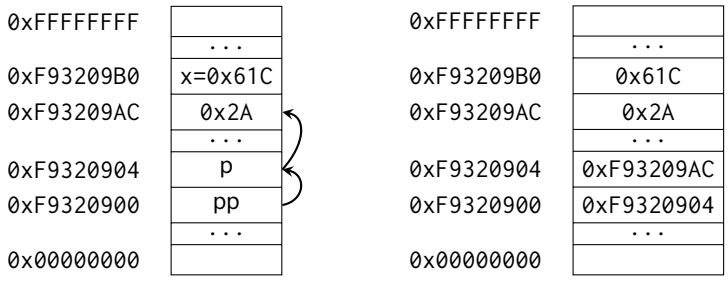
2 C

C is syntactically similar to Java, but there are a few key differences:

1. C is function-oriented, not object-oriented; there are no objects.
2. C does not automatically handle memory for you.
   - Stack memory, or *things that are not manually allocated*: data is garbage immediately after the *function in which it was defined* returns.
   - Heap memory, or *things allocated with* `malloc`, `calloc`, *or* `realloc`: data is freed only when the programmer explicitly frees it!
   - There are two other sections of memory that we learn about in this course, *static* and *code*, but we'll get to those later.
   - In any case, allocated memory always holds garbage until it is initialized!
3. C uses pointers explicitly. If `p` is a pointer, then `*p` tells us to use the value that `p` points to, rather than the value of `p`, and `&x` gives the address of `x` rather than the value of `x`.

   On the left is the memory represented as a box-and-pointer diagram.

   On the right, we see how the memory is really represented in the computer.



   Let's assume that **`int*`** `p` is located at `0xF9320904` and **`int`** `x` is located at `0xF93209B0`. As we can observe:

   - `*p` evaluates to `0x2A` (42_{10}).
   - `p` evaluates to `0xF93209AC`.
   - `x` evaluates to `0x61C`.
   - `&x` evaluates to `0xF93209B0`.

   Let's say we have an **`int`** `**pp` that is located at `0xF9320900`.

2.1 What does `pp` evaluate to? How about `*pp`? What about `**pp`?

2.2 The following functions are syntactically-correct C, but written in an incomprehensible style. Describe the behavior of each function in plain English.

(a) Recall that the ternary operator evaluates the condition before the `?` and returns the value before the colon (`:`) if true, or the value after it if false.

```c
int foo(int *arr, size_t n) {
    return n ? arr[0] + foo(arr + 1, n - 1) : 0;
}
```

(b) Recall that the negation operator, `!`, returns 0 if the value is non-zero, and 1 if the value is 0. The `~` operator performs a *bitwise not* (NOT) operation.

```c
int bar(int *arr, size_t n) {
    int sum = 0, i;
    for (i = n; i > 0; i--)
        sum += !arr[i - 1];
    return ~sum + 1;
}
```

(c) Recall that `^` is the *bitwise exclusive-or* (XOR) operator.

```c
void baz(int x, int y) {
    x = x ^ y;
    y = x ^ y;
    x = x ^ y;
}
```

(d) (Bonus: How do you write the *bitwise exclusive-nor* (XNOR) operator in C?)

3 Pointer Arithmetic

3.1 Consider the following blocks of C code:

```c
void printall(int *x) {
    // Suppose x points to 0xABDE2464
    const int NUM_ELEMS = 3;
    for(int i = 0; i < NUM_ELEMS; i += 1) {
        printf("Address: %x \n", x);
        x++;
    }
}
```

(a) What three memory addresses are printed by this program?

```c
void printall(char *x) {
    // Suppose x points to 0xABDE2464
    const int NUM_ELEMS = 3;
    for(int i = 0; i < NUM_ELEMS; i += 1) {
        printf("Address: %x \n", x);
        x++;
    }
}
```

(b) What three memory addresses are printed by this program?

4 Programming with Pointers

4.1 Implement the following functions so that they work as described.

(a) Swap the value of two **int**s. *Remain swapped after returning from this function.*
Hint: Our answer is around three lines long.

```
void swap(_________________, _________________) {
```

(b) Return the number of bytes in a string. *Do not use* `strlen`.
Hint: Our answer is around 5 lines long.

```
int mystrlen(_________________) {
```

4.2 The following functions may contain logic or syntax errors. Find and correct them.

(a) Returns the sum of all the elements in `summands`.

```
1  int sum(int *summands) {
2      int sum = 0;
3      for (int i = 0; i < sizeof(summands); i++)
4          sum += *(summands + i);
5      return sum;
6  }
```

(b) Increments all of the letters in the `string` which is stored at the front of an array of arbitrary length, `n >= strlen(string)`. Does not modify any other parts of the array's memory.

```
1  void increment(char *string, int n) {
2      for (int i = 0; i < n; i++)
3          *(string + i)++;
4  }
```

(c) Copies the string `src` to `dst`.

```
1  void copy(char *src, char *dst) {
2      while (*dst++ = *src++);
3  }
```

(d) Overwrites an input string `src` with "61C is awesome!" if there's room. Does nothing if there is not. Assume that `length` correctly represents the length of `src`.

```
1  void cs61c(char *src, size_t length) {
2      char *srcptr, replaceptr;
3      char replacement[16] = "61C is awesome!";
4      srcptr = src;
5      replaceptr = replacement;
6      if (length >= 16) {
7          for (int i = 0; i < 16; i++)
8              *srcptr++ = *replaceptr++;
9      }
10 }
```