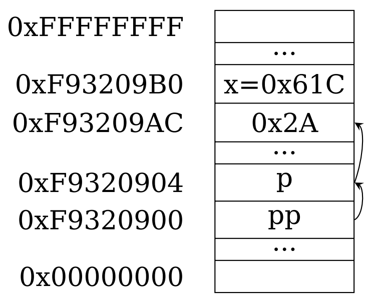
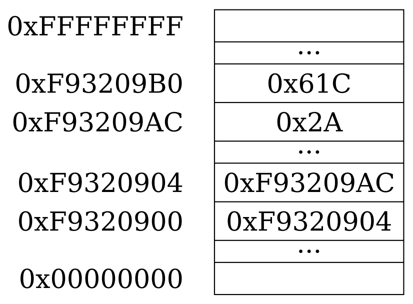
CS61C Spring 2018 Discussion 1 – Programming in C

1 C Introduction

C is syntactically very similar to Java, but there are a few key differences of which to be wary:

- C is function oriented, not object oriented. There are no objects.
- C does not automatically handle memory for you.
  - In the case of stack memory (things allocated in the "usual" way), a datum is garbage immediately after the function in which it was defined returns.
  - In the case of heap memory (things allocated with `malloc` and friends), data is freed only when the programmer explicitly frees it (by calling `free`).
  - In any case, allocated memory always holds garbage until it is initialized. **DO NOT** assume memory is zero'd unless `calloc` is used.
  - To specify the number of bytes required for a specific data type, use `sizeof`, e.g. to allocate space for 4 ints, do `malloc(sizeof(int)*4)`
- C uses pointers explicitly. `*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`. See the following example (the following addresses were chosen arbitrarily). On the left we see a diagram of pointers and memory that may help you visualize pointers. On the right, we see how those "boxes and arrows" are really represented.





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

  - *p should return 0x2A (42_{10}).
  - p should return 0xF93209AC.
  - x should return 0x61C.
  - &x should return 0xF93209B0.

Let's say we have an int **pp that is located at 0xF9320900. What would pp return? How about *pp? What about **pp?

- Pointer arithmetic: by now you should realize that pointers are nothing but just variables that hold addresses instead of data, which means we could add/subtract to pointers just as we do on normal int-valued variable. **Caveat:** syntactically, we use p+1 or p++ to move pointers, but you should understand the number of bytes added to the pointer depends on the type of the pointer. For example, if you have `int* p`, p++ will automatically add 4 bytes to p whereas if you have `char* p`, p++ will only add 1 byte. This is handled by the C compiler under the hood.

There are many other differences in C of which you should be aware of, but this should be enough for you to get your feet wet.

2 Uncommented Code? Yuck!

The following functions work correctly (note: this does not mean intelligently), but have no comments. Document the code to prevent it from causing further confusion.

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

2.
```
/*
 *
 */
int bar(int *arr, size_t n) {
    int sum = 0, i;

    for (i = n; i > 0; i--) {
        sum += !arr[i - 1];
    }

    return sum;
}
```

3.
```
/*
 *
 */
void baz(int x, int y) {
    x = x ^ y;
    y = x ^ y;
    x = x ^ y;
}
```

3 Programming with Pointers

Implement the following functions so that they perform as described in the comments.

1.
```
/* Swaps the value of two ints outside of this function. */
void swap(                     ) {





}
```

2.
```
/* Returns the number of characters in a string. Does not use strlen. */
int mystrlen(                     ) {




}
```

4 Problem?

The following code segments may contain logic and syntax errors. Find and correct them.

1.
```
/* Returns the sum of all the elements in SUMMANDS. */
int sum(int* summands) {
    int sum = 0;
    for (int i = 0; i < sizeof(summands); i++) // ask yourself what would sizeof(summands) return
        sum += *(summands + i);
    return sum;
}
```

2.
```
/* Increments all the letters in the string STRING, held in an array of length N.
 * Does not modify any other memory which has been previously allocated. */
void increment(char* string, int n) {
    for (int i = 0; i < n; i++)
        *(string + i)++;
        // p++ is syntactic sugar for p = p + 1
}
```

3.
```
/* Overwrites an inputted string with ''61C is awesome!'' if there's room.
 * Does nothing if there is not. Assume that srcLength correctly represents
 * the length of src. */
void CS61C(char* src, size_t srcLength) {
    char *srcptr, replaceptr;
    char replacement[16] = ``61C is awesome!'';
    replaceptr = replacement;
    if (srcLength >= 16) {
        for (int i = 0; i < 16; i++)
            *src++ = *replaceptr++;
    }
}
```