CS 11 C track: lecture 4

- Last week: arrays

- This week:

  - Recursion

  - Introduction to pointers

Lab 4

- Harder than previous labs
- One non-obvious trick
    - hints on web page
    - email me if get stuck
- Support code supplied for you
- Read carefully!

Recursion (1)

- Should be familiar from CS 1

- Recursive functions call themselves

- Useful for problems that can be decomposed

  in terms of smaller versions of themselves

Recursion (2)

```c
int factorial(int n) {

    assert(n >= 0);

    if (n == 0) {

        return 1;   /* Base case. */

    } else {

        /* Recursive step: */

        return n * factorial(n - 1);

    }

}
```

Recursion (3)

```
factorial(5)

--> 5 * factorial(4)

--> 5 * 4 * factorial(3)

--> 5 * 4 * 3 * factorial(2)

--> 5 * 4 * 3 * 2 * factorial(1)

--> 5 * 4 * 3 * 2 * 1 * factorial(0)

--> 5 * 4 * 3 * 2 * 1 * 1

--> 120
```

Pointers (1)

- Address:
  - A *location* in memory where data can be stored
  - *e.g.* a variable or an array
  - Address of variable **x** is written **&x**
- Pointer:
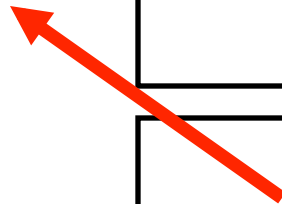 - A variable which holds an address

Pointers (2)

name	address	contents
i	**0x123aa8**	**10**
j	**0x123aab**	**0x123aa8**

```
int  i = 10;
int *j = &i;   /* j "points" to i */
```

Pointers (3)

```
int  i = 10;

int *j = &i;

printf("i = %d\n", i);

printf("j = %x\n", j);

printf("j points to: %d\n", *j);
```

Pointers (4)

- `&i` is the address of variable `i`

- `*j` is the contents of the address stored in pointer variable `j`

    - *i.e.* what `j` points to

- `*` operator dereferences the pointer `j`

Pointers (5)

- The many meanings of the * operator:

  - Multiplication

    ```
    a =  b * c;
    ```

  - Declaring a pointer variable

    ```
    int *a;
    ```

  - Dereferencing a pointer

    ```
    printf("%d", *a);
    ```

Pointer pitfalls (1)

- Declaring multiple pointer variables:

`int *a, *b;   /* a, b are ptrs to int */`

- If you do this:

`int *a, b;    /* b is just an int */`

- Then only the first variable will be a pointer

- Rule: every pointer variable in declaration must be preceded by a `*`

Pointer pitfalls (2)

- Note that

```
int *j = &i;
```

- really means

```
int *j; /* j is a pointer to int */

j = &i; /* assign i's addr to j */
```

- Don't confuse this `*j` with a dereference!

Pointers (6)

- A harder problem:

```
int   i = 10;
int  *j = &i;
int **k = &j;
printf("%x\t%d\n", &i, i);
printf("%x\t%x\t%d\n", &j, j, *j);
printf("%x\t%x\t%x\t%d\n",
        &k, k, *k, **k);
```

Pointers (7)

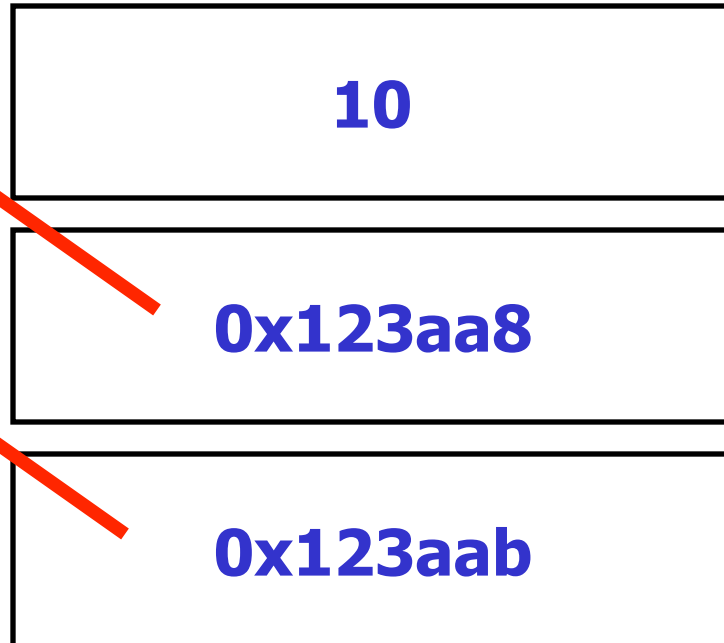| name	address	contents
i	**0x123aa8**	**10**
j	**0x123aab**	**0x123aa8**
k	**0x123ab0**	**0x123aab**

Assigning to pointers (1)

```c
int i = 10;
int *j = &i;
int *k;
/* Assign to what j points to: */
*j = 20;   /* Now i is 20. */
/* Assign j to k: */
k = j;     /* Now k points to i too. */
/* Assign to what j points to: */
*j = *k + i;  /* Now i is 40. */
```

Assigning to pointers (2)

- When pointer variable is on left-hand side of an assignment statement, what happens depends on whether it's dereferenced or not

  - no dereference: assign the value on RHS (an address) to the pointer variable on the LHS

```
j = k;
```

  - dereference: assign value on RHS into location corresponding to where pointer points to

```
*j = *k + 10;
```

Assigning to pointers (3)

- When pointer variable is declared and assigned to on the same line:

`int *j = k;`

- it means:

`int *j;   /* declare j    */`

`j = k;    /* assign to j */`

- *i.e.* assign the value on RHS (an address) to the pointer variable on the LHS

Mnemonics: fetch/store

- When you use the * (dereference) operator in an expression, you _fetch_ the contents at that address

```
printf("j's contents are: %d\n", *j);
```

- When you use the * (dereference) operator on the left-hand side of the = sign in an assignment statement, you _store_ into that address

```
*j = 42;    /* store 42 into address */
```

Pointers – call by reference (1)

- Can use pointers for a non-obvious trick
- Recall: in C, variables are copied before being sent to a function
  - referred to as "call-by-value"
- Significance is that passing a variable to a function *cannot* change the variable's value
- But sometimes we *want* to change the variable's value when function returns

Pointers – call by reference (2)

```c
void incr(int i) {
    i++;
}
/* ... later ... */
int j = 10;
incr(j); /* want to increment j */
/* What is j now? */
/* Still 10 – incr() does nothing. */
```

Pointers – call by reference (3)

```c
void incr(int *i) {
    (*i)++;
}
/* ... later ... */
int j = 10;
incr(&j);
/* What is j now? */
/* Yep, it's 11. */
```

Pointers – call by reference (4)

```
int j = 10;

incr(&j);
```

- You should be able to work out why this works

- Where have we seen this before?

```
int i;

scanf("%d", &i); /* read in i */
```

Pointers – call by reference (5)

- Easy mistake to make:

```
void incr(int *i) {
    *i++;   /* Won't work! */
    /* Parsed as: *(i++); */
}
```

- Need to say `(*i)++` here
- Precedence rules again; use parens `()` if any confusion can exist

Next week

- Pointers and arrays

  (the untold story)

- Dynamic memory allocation