Day 13 (Mon 02/21)

- exercise 12 review
- day 13 recap questions
- exercise 13

Announcements/reminders:

- HW3 due Friday, 2/25 by 11pm
- HW4: soon

Exercise 12 review

Declaration of search function:

How it is called:

```
pos = search(arr1, arr1 + 10, 318);
```

Declaration:

```
int *search(int *start, int *end, int searchval);
```

Exercise 12 review (continued):

Useful property when lower bound of search range is inclusive, and upper bound is exclusive: end - start is the number of elements in the range

So:

```
int *search(int *start, int *end, int searchval) {
  int num_elts = (int) (end - start);
  if (num_elts < 1) {
    return NULL; // no elements in range
  } else {
    // general case: check middle element, if it's equal to
    // searchval, success, otherwise continue recursively on
    // left or right side of range
  }
}
```

Exercise 12 review (continued):

```
// search, general case
int *mid = start + (num_elts/2);
if (*mid == searchval) {
  return mid; // success, found the search value
} else if (*mid < searchval) {
  // continue recursively in right side of range
} else {
  // continue recursively in left side of range
}
```

Exercise 12 review (continued):

```
// in the test code, finding the index of the matching element
pos = search(arr1, arr1 + 10, 318);
assert(pos != NULL);
assert(*pos == 318);
// TODO: compute the index of the matching element
index = pos - arr1;  // <-- add this
assert(2 == index);
```

Exercise 12 review (continued):

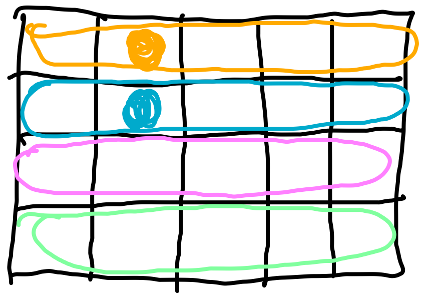
general observation about 2-D arrays: if p is a pointer to an element, and N is the number of columns in one row, then

p + N

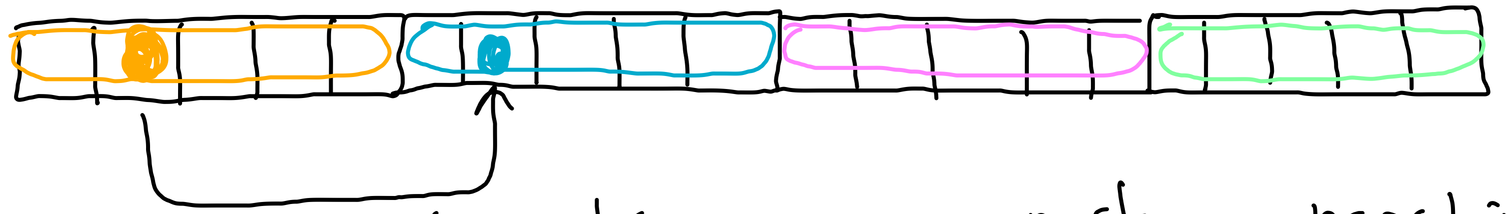
yields a pointer to an element that is in the same column and next row from the element p points to

4 x 5 2-D array

conceptually:



as laid out in memory:



elements one row apart: separation is same as # of elements per row (i.e., number of columns)

Exercise 12 review (continued):

makeCol -

```
// TODO: declare the unit variable (array of 9 integers, to be returned)
int *unit = malloc(9 * sizeof(int));
```

makeCube -

```
// TODO: declare the unit variable (array of 9 integers, to be returned
int *unit = malloc(9 * sizeof(int));
```

checkRows -

```
// TODO: call check on current row and add to variable good
good += check(&table[r][0]);
```

pass address of first element in row:
the elements of the row are contiguous
in memory

observation: elements in a single row are contiguous in memory
(each row of a 2-D array can be treated as a 1-D array)

Exercise 12 review (continued):

```
checkCols -
  for (int c = 0; c < SIZE; c++) {
    // TODO: call makeCol on current column and assign result to column
    column = makeCol(&table[0][c]);     // <-- get one column of values
    good += check(column);
    free(column);                       // <-- free dynamic array
  }
```

address of "top" element in column c

```
checkCubes -

  // TODO: call makeCube on current cube and assign result to variable cube
  cube = makeCube(&table[r][c]);    // <-- get 3x3 "cube" of values
  good += check(cube);
  free(cube);                       // <-- free dynamic array
```

address of upper-left element of "cube"

Exercise 12 review (continued):

main (in sudoku.c) -

code does not call fclose to close input file: should modify main function so that infile is guaranteed to be closed (using fclose) if it is opened successfully

Makefile

CFLAGS should include the -g option (to enable debug symbols)

running valgrind:

```
valgrind ./main --leak-check=full --show-leak-kinds=all <name of input file>
```

will show unclosed files as memory leaks

Day 13 recap questions:

1. What is struct in C?
2. How are the fields of a struct passed into a function - by value or by reference?
3. What is the size of a struct? What is structure padding in C?
4. What is the difference between lifetime and scope of a variable?
5. What is variable shadowing (i.e. hiding)?
6. What is the output of the below program?

1. struct introduces a *used-defined data type*

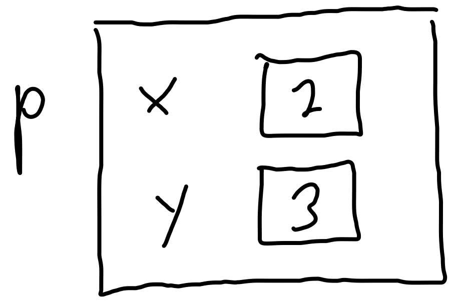
Very much like a class in Java or Python, but with only the ability to include member variables, not member functions.

An instance of a struct is a "bundle" of variables that are packaged as a single entity.

Example:

```
struct Point {
  int x, y;
};

// ... elsewhere in the program ...
struct Point p = { .x = 2, .y = 3 };
```



} struct instance is a "bundle" of variables; a.k.a., an <u>object</u>

2. Instances of a struct type are passed by value. E.g.

```
struct Point { int x, y; };

void f(struct Point p, int dx) {
  p.x += dx;
}

int main(void) {
  struct Point q = { .x = 4, .y = 5 };
  f(q);
  printf("%d,%d\n", q.x, q.y); // prints "4,5"
  return 0;
}
```

Trace:

f

main

3. sizeof(struct Foo) is

the sum of the sizes of the fields of struct Foo, plus the total size of any padding inserted by the compiler to ensure that fields are correctly aligned

alignment: the memory address of a variable (including a field variable in an instance of a struct type) must be a multiple of the size of the field.

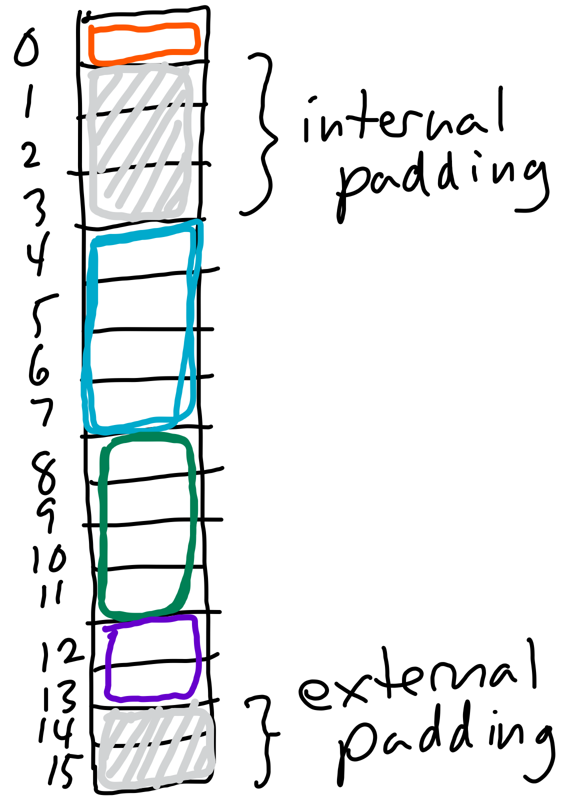
E.g., a 4-byte int variable (or struct field) must have its storage allocated starting at a machine address that is a multiple of 4.

The compiler will insert padding automatically: you don't need to do anything special. sizeof(struct Foo) will always take the padding into account. Just trust that the compiler will figure out the right struct layout to use.

Example:

```
struct Player {
    char symbol;
    int x, y;
    short health;
};
```

Possible memory layout:



4. Lifetime: the interval from (1) the point in time when a variable is created, to (2) the point in time when a variable is destroyed.

Lifetime of a variable is a dynamic property of the program.

(determined when the program executes

Examples:

the lifetime of a local variable is the duration of the function call

the lifetime of a global variable is the duration of the entire program

Scope: the region of the program code in which a variable may be accessed

Scope of a variable is a static property of the program.

(known when the program is compiled

Examples:

the scope of a local variable is from its declaration to the closing "}" of the block in which it's defined

the scope of a global variable is the entire program (assuming that there is a declaration or definition of the variable in the current block, or in the enclosing block

5. Shadowing: a variable declaration in a nested scope has the same name as a variable in an "outer" scope.

E.g.

```
int x;

void foo(int x) {
  {
    int x = 5;
    printf("%d\n", x); // prints "5"
  }
  printf("%d\n", x); // prints "4"
}

int main(void) {
  x = 3;
  foo(4);
  printf("%d\n", x); // prints "3"
  return 0;
}
```

6.

```
#include <stdio.h>
int foo;
void bar() {
  int foo = 3;
  {
    extern int foo;
    printf("%d; ", foo);
    foo = 2;
  }
  printf("%d; ", foo);
}
void baz() { printf("%d; ", foo); }
int main() {
  {
    int foo = 5;
    bar();
    printf("%d; ", foo);
  }
  baz();
  return 0;
}
```