601.220 Intermediate Programming

Spring 2023, Day 13 (February 20)

Today's agenda

- Exercise 12 review
- Lifetime/scope, struct types, random number generation
- Exercise 13

Reminders/Announcements

- → HW3 due Friday (Feb 24th)
  - This is a challenging assignment, don't wait until the last minute
- Midterm project team registration: soon

Exercise 12 review

Declaration of search function:

How it is called:

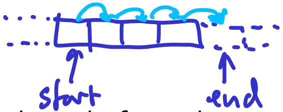
```
pos = search(arr1, arr1 + 10, 318);
```

Declaration:

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

inclusive exclusive

Exercise 12 review



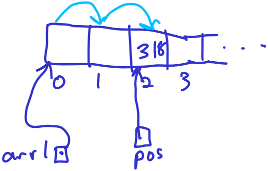
Useful property when lower bound of search range is inclusive, and upper bound is exclusive: `end - start` is the number of elements in the range. So:

```
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

```c
// 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



```
// 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);
```

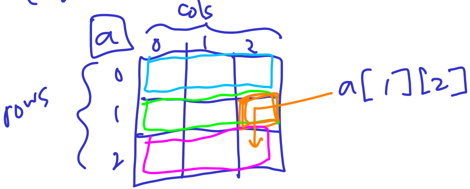
Exercise 12 review

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

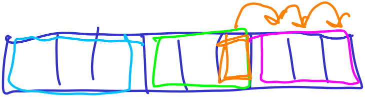
```
p + N
```

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

e.g. 3 x 3 (N = 3)



as 1-D array



Exercise 12 review

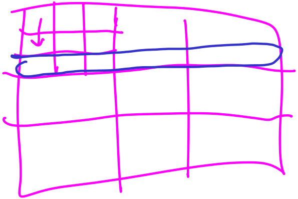
`makeCol`:

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

Exercise 12 review

`makeCube`:

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



Exercise 12 review

`checkRows`:

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

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

`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
}
```

Exercise 12 review

`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
```

Exercise 12 review

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>
```

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. What is *struct* in C?

`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 };
```

2. How are the fields of a struct passed into a function - by value or by reference?

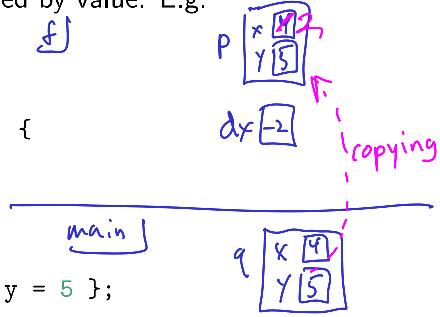
Instances of a struct type are passed by value. E.g.

```c
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, -2);
  printf("%d,%d\n", q.x, q.y); // prints "4,5"
  return 0;
}
```

4 5



3. What is the size of a *struct*? What is structure padding in C?

`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.

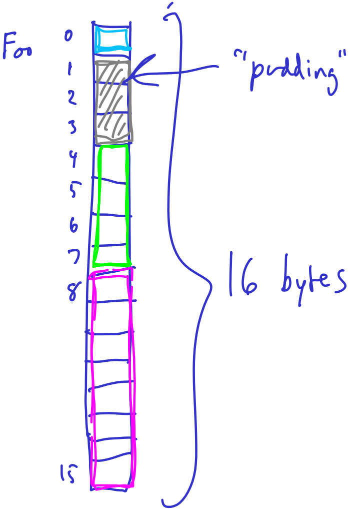
The compiler will insert padding automatically: you don't need to do anything special. `sizeof(struct Foo)` will always take the padding into account. Just trust that the compiler will figure out the right struct layout to use.

struct padding example

```
struct Foo {
  char a;
  int b;
  long c;
};

// ...

struct Foo f;
printf("%lu\n", sizeof(f));
```



4. What is the difference between lifetime and scope of a variable?

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. Examples:

dynamic

- 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

static

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

“block” { .... }

- 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. What is variable shadowing (i.e. hiding)?

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

Shadowing example

```
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;
}
```

foo

x (param) 4

x (block) 5

main

x 3

6. What is the output of the below program?

```c
#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;
}
```

global

same

`.` vs. `->`

To access a member variable of a `struct` instance directly, use the "`.`" operator. To access a member variable of a `struct` instance indirectly via a pointer, use the `->` operator.

Note that `p->x` means exactly the same thing as `(*p).x`. It's just a more convenient syntax.

x.f if x IS instance of struct type

x->f if x points to instance of struct

Example of . vs. ->

```
struct Player { int x; int y; int health; };

struct Player player;

player.x = 42;
player.y = 17;

struct Player *p = &player;
p->health = 100;
```

Exercise 13

- Working with `struct` types, including pointers to instances of `struct` types
- Talk to us if you have questions!

Notes

Notes

Notes

Notes

Notes