switch statements

A switch statement is a different way of expressing a conditional.

The general format of this looks like:

```c
switch (e) {
  case c1:
    // do something
    break;
  case c2:
    // do something else
    break;
  // ...
  default:
    // do something in the default case
    break;
}
```

Each ci should evaluate to a constant integer type (this can be of any size, so chars, ints, long long ints, etc).

For example, consider this function that moves on a board. It takes direction ('l', 'r', 'u', or 'd') and prints an English description of the direction.

```c
void print_dir(char c) {
  switch (c) {
    case 'l':
      printf("Left\n");
      break;
    case 'r':
      printf("Right\n");
      break;
    case 'u':
      printf("Up\n");
      break;
    case 'd':
      printf("Down\n");
      break;
    default:
      fprintf(stderr, "Specify a valid direction!\n");
      break;
  }
}
```

The break statements here are important: If we don't have them, we get fall-through, which is often useful, but can lead to unanticipated results.
Here’s some code that takes a positive number at most 10 and determines whether it is a perfect square:

```c
int is_perfect_square(int x) {
    REQUIRES(1 <= x && x <= 10);
    switch (x) {
    case 1:  
        case 4: 
        case 9: 
            return 1;
            break;
    default: 
        return 0;
        break;
    }
}
```

The behavior here is called fall-through. If misused, this can lead to confusing behavior. For example, let’s look at the following buggy code and its output.

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    if (argc > 1) {
        int a = atoi(argv[1]);

        switch (a % 2) {
        case 0: 
            printf("x is even!\n");
            default: 
                printf("x is odd!\n");
        }
    }
    return 0;
}
```

There are two cases: when the input is odd and when it is even. Let’s look at both of them.

```
$ ./badswitch 1
x is odd!
$ ./badswitch 2
x is even!
```

There are some details about variables local to the body of one case that we’ll give you with your next homework. They’re relatively simple so we won’t talk about them now.
structs that aren’t pointers

We’ve almost always used pointers to structs previously in this class.

We can also just use structs, without the pointer. We set a field of a struct with dot-notation, as follows:

typedef struct list_node node;
struct list_node {
    struct list_node *next;
    void *data;
};
// ...
node a;
a.data = NULL;
a.next = NULL;

The notation we’ve used throughout the semester to access a field of a pointer to a struct is p->f. This is just syntactic sugar for (*p).f.

Flexible array members of structs

If we want a variable length array as part of a struct, we need to use the C99 standard. It lets us declare structs as follows:

struct c0_array {
    int count;
    int elt_size;
    char elems[];
}

If we call sizeof(struct c0_array), it gives us the size of the array assuming that elems has length 0.

Thus, to allocate this struct, we also need to be careful to request a specific size for the array. If we want to have elems be size 42, we could do:

struct c0_array *a = xalloc(1, sizeof(struct c0_array) + 42);
a -> count = 42;
a -> elt_size = 1;

Be careful with structs in C. Due to a principle called alignment (or padding), sizeof(struct foo) is not necessarily equal to the sum of the sizes of the elements of foo — it could also be larger.

You’ll learn more about this in 15-213, but for now it suffices to know that if you are determining the size of a struct, you should use sizeof.

Arguments to macros

We can define macros that take arguments. These simply do a textual find-and-replace when they are “called”.

This can lead to many problems.

For instance, consider this definition of a macro:
#define MAX(a,b) a > b ? a : b

What happens if we call 5 * MAX(4 + 3, 10)?

Well, we just do simple replacement, so this is expanded to 5 * 4 + 3 > 10 ? 4 + 3 : 10.

This does not run the way we want it to: We need parentheses. Around everything.

#define MAX(a, b) ((a) > (b) ? (a) : (b))

This gets us: 5 * ((4 + 3) > (10) ? (4 + 3) : 10), which works much more reliably.

It’s also very important to never have a statement with side effects in the arguments to a macro:

MAX(x++, y++); will expand to:

((x++) > (y++) ? (x++ : (y++)). This is bad, because one of x++ and y++ will be evaluated twice. Moral of the story? Avoid calling macros with expressions that have side effects.

inline functions provide a way to avoid these problems and to get many of the performance benefits that macros provide. If you’re interested in more details, you can ask me about it at some point or look up more information online, but we won’t cover it now due to the fact that this recitation is not long enough.

**Casting pointers to ints and signed to unsigned**

Casting from pointers to integers and signed values to unsigned values is implementation-defined in C. (That is, C does not mandate the way that compilers should handle these details. For homework 8, we’ll use the behaviors that GCC defines.)

A few details:


In assignment 8, we’ll provide you with INT(p) and VAL(x) to cast between integers and pointers. You can look at their definitions to see how they work.