Everything has an address!

Well, anything you can name—all variables and functions.

We can use the address of operator, & , to find what this address is.

This is useful if we want to modify a variable in place.

Checkpoint 0

```c
#include <stdio.h>
#include "contracts.h"

void bad_mult_by_2(int x) {
    x = x * 2;
}

void mult_by_2(int* x) {
    REQUIRES(x != NULL);
    *x = *x * 2;
}

int main () {
    int a = 4;
    bad_mult_by_2(a);
    printf("%d\n", a);
    mult_by_2(&a);
    printf("%d\n", a);
    return 0;
}
```

What is the output when this code is run? Why?

**Solution:** The output is

The reason that mult_by_2 works and the other function doesn’t is that C passes a copy of its arguments in to the function it calls, so if you directly modify your argument you’re only modifying your copy of it. In mult_by_2, we’re given the address where a variable is stored and we go there and update it.

**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:
        ...
}
```
Each ci should evaluate to a constant integer type (this can be of any size, so chars, ints, 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
");
        break;
    case 'r':
        printf("Right
");
        break;
    case 'u':
        printf("Up
");
        break;
    case 'd':
        printf("Down
");
        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. The behavior here is called fall-through.

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

**Checkpoint 1**

```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;

What's wrong with this code? How would you fix it?

Solution: 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!
x is odd!

The code is missing an essential break statement at the end of the first case.

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:

```c
#define ARRAY_LENGTH 10
struct point {
    int x;
    int y;
};
int main () {
    struct point a;
    a.x = 3;
    a.y = 4;
    struct point* arr = xmalloc(ARRAY_LENGTH * sizeof(struct point));
    // Initialize the points to be on a line with slope 1
    for (int i = 0; i < ARRAY_LENGTH; i++) {
        arr[i].x = i;
        arr[i].y = i;
    }
}
```

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.

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 Lab 9, we'll use...
the behaviors that GCC defines.)

A few details:

The GCC documentation specifies how casting from pointers to ints works:
http://gcc.gnu.org/onlinedocs/gcc-4.3.5/gcc/Arrays-and-pointers-implementation.html#Arrays-and-pointers-implementation

In Lab 9 (the C0 Virtual Machine), we’ll provide you with INT(p) and VAL(x) to cast between integers and pointers.

Make sure to review the lecture notes for more details on casting.

Checkpoint 2

What’s wrong with each of these pieces of code?

```c
1 // We're returning the address of a local variable, so the value of x
2 // may change unexpectedly if another function's local variables are in the
3 // wrong place on the stack.
4 int *add_dumb(int a, int b) {
5    int x = a + b;
6    return &x;
7 }

1 // A + i implicitly actually adds i * sizeof(int) to A. This is so that
2 // A[i] is the same as *(A + i).
3 // The loop upper bound should just be 10.
4 int main () {
5    int *A = xcalloc(10, sizeof(int));
6    for (int i = 0; i < 10 * sizeof(int); i++) {
7        *(A + i) = 0;
8    }
9    free(A);
10   return 0;
11 }

1 // When we call a function we make a copy of the argument, so add_one is
2 // modifying its own copy and the copy in main is unchanged.
3 void add_one(int a) {
4    a = a + 1;
5 }
6 int main() {
7    int x = 1;
8    add_one(x);
9    printf("%d\n", x);
10   return 0;
11 }

1 // "x = 1" assigns x to 1. In C, the expression evaluates to 1, since that's
2 // what we assigned x to. Any non-zero value is true, so we print out "woo\n"
3 int main() {
4    int x = 0;
5    if (x = 1) {
6        printf("woo\n");
7    }
8    return 0;
9 }
```
// s is not properly NUL-terminated, so when we print the string we have
// undefined behavior
int main() {
    char s[] = {'a', 'b', 'c'};
    printf("%s\n", s);
    return 0;
}

// strlen(y) doesn’t count the null-terminator, so we copy at most 6 characters
// into x. Since malloc doesn’t intialize memory to 0, this means that x
// might not be NUL-terminated, and so trying to get the length of x is
// undefined behavior
int main() {
    char *y = "hello!";
    char *x = xmalloc(7 * sizeof(char));
    strncpy(x, y, strlen(y));
    printf("%zu\n", strlen(x));
    free(x);
    return 0;
}

// We’re freeing something we never malloc’d, which we should never do — only
// free something you malloc’d. This gives undefined behavior as-is.
int foo(char *s) {
    printf("The string is %s\n", s);
    free(s);
}
int main() {
    char *s = "hello";
    foo(s);
    return 0;
}