Everywhere 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.

For instance, if we want a function that doubles a given `int` and modifies it in place, we need to take a pointer to that `int`, since C is pass by value:

```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;
}
```

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 the second example, 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:
        // 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
1 void print_dir(char c) {
2     switch (c) {
3         case 'l':
4             printf("Left
");
5             break;
6         case 'r':
7             printf("Right
");
8             break;
9         case 'u':
10            printf("Up
");
11            break;
12         case 'd':
13            printf("Down
");
14            break;
15         default:
16             fprintf(stderr, "Specify a valid direction!\n");
17             break;
18     }
19 }
```

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
1 int is_perfect_square(int x) {
2     REQUIRE(1 <= x && x <= 10);
3     switch (x) {
4         case 1:
5             return 1;
6         case 4:
7             return 1;
8         case 9:
9             return 1;
10            break;
11         default:
12             return 0;
13            break;
14     }
15 }
```

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
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(int argc, char **argv) {
4     if (argc > 1) {
5         int a = atoi(argv[1]);
6         switch (a % 2) {
7             case 0:
8                 printf("x is even!\n");
9             default:
10                 printf("x is odd!\n");
11             }
12     return 0;
13 }
14 return 0;
15 }
```
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!

Note that there are some details about variables local to the body of one case that we’ll give you with homework 8.

**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 homework 8, we’ll use the behaviors that GCC defines.)

A few details:

The GCC documentation specifies how casting from pointers to ints works:


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.

Make sure to review [http://www.cs.cmu.edu/~rjsimon/15122-s13/20-types.pdf](http://www.cs.cmu.edu/~rjsimon/15122-s13/20-types.pdf) for more details on casting.
What's wrong with this code?

```c
1 int *add_dumb(int a, int b) {
2     int x = a + b;
3     return &x;
4 }

1 int main () {
2     int *A = xcalloc(10, sizeof(int));
3     for (int i = 0; i < 10 * sizeof(int); i++) {
4         *(A + i) = 0;
5     }
6     free(A);
7     return 0;
8 }

1 void add_one(int a) {
2     a = a + 1;
3 }
4 int main() {
5     int x = 1;
6     add_one(x);
7     printf("%d\n", x);
8     return 0;
9 }

1 int main() {
2     int x = 0;
3     if (x = 1) {
4         printf("woo\n");
5     }
6     return 0;
7 }

1 int main() {
2     char s[] = {'a', 'b', 'c'};
3     printf("%s\n", s);
4     return 0;
5 }

1 int main () {
2     char *y = "hello!";
3     char *x = xmalloc(7 * sizeof(char));
4     strncpy(x, y, strlen(y));
5     printf("%zu\n", strlen(x));
6     free(x);
7     return 0;
8 }

1 int foo(char *s) {
2     printf("The string is %s\n", s);
3     free(s);
4 }
5 int main() {
6     char *s = "hello";
7     foo(s);
8 }
```
return 0;
}