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

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

switch statements

A switch statement is a different way of expressing a conditional. The general format of this looks like:

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.
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) {

    REQUIRE(1 <= x && x <= 10);
    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?
**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 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?

a)

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

c)  
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 }

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

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

f)  
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 }

g)  
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;
9 }