C Contracts: REQUIRES(), ENSURES(), ASSERT()

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

```
#include <stdio.h>
#include "lib/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?

switch statements

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

```
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. 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");
                break;
            default:
                printf("x is odd!\n");
                break;
        }
    }
    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 Lab 9, 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)
```c
int main () {
    int A = xcalloc(10, sizeof(int));
    for (int i = 0; i < 10 * sizeof(int); i++) {
        *(A + i) = 0;
    }
    free(A);
    return 0;
}
```

c)
```c
void add_one(int a) {
    a = a + 1;
}
```
```c
int main() {
    int x = 1;
    add_one(x);
    printf("%d\n", x);
    return 0;
}
```

d)
```c
int main() {
    int x = 0;
    if (x = 1)
        printf("woo\n");
    return 0;
}
```

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

f)
```c
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;
}
```

g)
```c
int foo(char* s) {
    printf("The string is %s\n", s);
    free(s);
} 
```
```c
int main() {
    char* s = "hello";
    foo(s);
    return 0;
}
```