Table of Contents

1 C Syntax Overview
   - Addresses and Referencing
   - Switch Statements
   - Structs by Value
   - Casting

2 Exercises
Everything has an address!

### Referencing

```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;
}
```

### Checkpoint 0

- In `bad_mult_by_2`, `x` is a copy of `a`, so assignment to `x` does not change `a`.
- However, in `mult_by_2`, `x` is a pointer to `a`. In that case, assignment to `x` still doesn’t change `a`; but assignment to `*x` does.
- This is an example of “Pass by value” versus “Pass by reference”. The final output is: 4 8
Switch Introduction

Syntax

```c
switch (e) {
    case c1:
        // do something
        break;
    case c2:
        // Fall through
    case c3:
        // do something
        break;
    default:
        // do something
        break;
}
```

Details

When a switch statement is encountered, control flow switches to the first case \( c \) such that \( e == c \). Control then continues down through the switch statement until a `break`.

\( e \) must evaluate to an integer type and the \( c \)'s must be constant integer types.

So 1, 10+5, 0xffff0001, and 'x' are acceptable but `get_integer_type()` and `some_var` are not.
Switch Pitfalls

Bad Switch

```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;
}
```

Checkpoint 1

What’s wrong with this code? How can we fix it?
When the input is even, we fall through and execute both print statements. We need a break statement after the first case.
#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;
    }
}
Structs that aren’t pointers

Pointers vs Values

What are some differences between structs and struct pointers?

- **Syntax:** `p.f` instead of `p->f`
- Don’t need to allocate or free memory
- Pass by value instead of pass by reference

Struct pointers can also be arrays of structs, by allocating a multiple of the size of the struct.
Casting between types can be a bit complicated. Overzealous casting can introduce subtle and strange bugs into your code. If you have to cast (and you will), remember these guidelines:

- Casting preserves bits as much as possible. So casting between signed and unsigned will generally NOT preserve value.
- Casting to a smaller type truncates the bits.
- Casting to a larger type sign-extends for signed types, but zero-extends for unsigned types.
- A pointer cast from an integer won’t point anywhere meaningful; don’t dereference it!

We will provide you with macros to cast between pointers and integers.
What’s Wrong with this Code?

Bad Code A

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

Checkpoint 2

x is on `add_dumb`'s stack, which is freed when it returns. `add_dumb` should instead take in a pointer, or just return an integer.
Bad Code B

```c
void main() {
    int* A = xcalloc(10, sizeof(int));
    for (int i=0; i < 10*sizeof(int); i++) {
        *(A + i) = 0;
    }
    free(A);
}
```

Checkpoint 2

Adding \( n \) to a pointer actually adds \( n \) times the size of that pointer. This will cause \( *(A + i) \) to point to outside the allocated array. The loop guard here should be \( i < 10 \).
What's Wrong with this Code?

Bad Code C

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

Checkpoint 2

a is passed by value, so `add_one` has no affect on `x`. `add_one` should take an integer pointer, and `main` should pass a reference to `x`. 
What’s Wrong with this Code?

Bad Code D

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

Checkpoint 2

`x = 1` assigns to `x` instead of testing its value. Use `==` instead.
What’s Wrong with this Code?

**Bad Code E**

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

**Checkpoint 2**

C strings must be null terminated. A ‘\0’ should be put at the end of s.
What’s Wrong with this Code?

**Bad Code 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;
}
```

**Checkpoint 2**

Once again, C strings must be nul terminated. Furthermore, malloc does not zero out allocated memory. This code copies only the six non-nul characters of y to x, resulting in x being non-nul terminated. We should copy `strlen(y)+1` characters.
What’s Wrong with this Code?

Bad Code G

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

Checkpoint 2

`s` was not mallocced and therefore should not be freed. Specifically, the "Hello World" literal is stored in read-only memory for the duration of this program, and cannot be modified (e.g., freed).