Lecture 7

1. Syntactic Traps

1.1. Multiple Affectations

The presence of different assignment operators in the same instruction

```
 char x = 'A';
char y;
y = x = getchar();
printf("%c", y);
```

will result in

```
 > B
B
```

So, the assignments took order from right to left. `getchar()` ==> `x` ==> `y`.

It's better to write it

```
 x = getchar();
y = x;
```

1.2. Relying on C Properties

When having different operators, use parantheses to define your order of the operations.

DON'T

```
 x = 1 / 2 * p;
```

As this code will result in 0 always as the `1/2` will be executing first resulting in 0 that will be multiplied with p to result in 0.

DO

```
 x = 1 / (2 * p);
```

Another example,

```
word = high << 8 + low;
```

The intention was that we have 2 registers `low` and `high` and the a 16 bit number is stored on these register the higher 8-bits are stored in `high` and the lower 8-bits in `low`. Then we combining all bits, the bits in `high` is left-shifted by 8 then added to the `low`. Will this happen ?

Answer: No, because in the precedence rules, addition will be done before shifting.

```
// What will happen...
word = high << (8 + low);
```

Solution:

```
word = (high << 8) + low;
```

1.3. Overusing of the Post and Pre Fix Operators

```
int x[10], *ptr = &x, y = 4;

*ptr++ = --y;
```

Can you guess the output from the first try? The overuse of increments and decrements without parantheses will produce a code that is hard in reading and maintanince.

1.4. Abusing Syntax

For example, when using for loops in C, the usual syntax is definition and intialization, condition, increment or decrement.

Unfollowing such conventions will result in unreadable code,

```
int i = 5;
for(; i--; x[i] = 0);
```

For the second statement in the loop, `i--`, `i` will be checked if it's true or false - true if `i` is any value other than 0 - then, `i` will be decremented. Then, `x[i] = 0` will be executed. When checking for `i` after it was decremented to 0, the loop will break.

Too much confusing that could be avoided if the code was

```
for(int i = 5; i > 0; i--)
{
    x[i] = 0;
}
```

1.5. Permissive Switch

When writing a `switch` statement, if you write set of instructions before the first `case`, these instructions will not be executed.

```
int x = 5;
switch(x)
{
    int y = 1;
    case 5:
        x = y;
        break;
    default:
        break;
}
printf("%d", x);
```

When running the above code in my CodeBlocks, it resulted in an output of `0`. With Eng. Hossam, the result was `8`.

The code will not suffer from a compilation error becuase the **declaration** of `y` is not done throught the executable code but the **assignment** of `y` is done in the executable code.

1.6. Semi-Colon Usage

Empty semi-colon is an instruction of its own. If added after `if` statement or a `for` loop that don't have braces, the `if` statement (or `for` loop) will execute nothing.

```
 if(x > 0);
    x = 0;
printf("%d", x);
```

- If `x` is +ve, the output will be 0.
- If `x` is -ve or 0, the output will still be 0.

Because, the assignment of `x` to 0 doesn't depend on the `if` statement if it was true or false as if it was true, it will execute only the first instruction as, it doesn't have brackets, which the empty semi-colon.

Exactly the same for

```
 if(x > 0);
{
    x = 0;
}
printf("%d", x);
```

Adding an empty semi-colon is a bad practice, missing one is also bad.

```
 int function(void){
    int x = 5;
    if (x > 0)
        return
    x = 1;
}

int main()
{
    int  y = function();
    printf("%d", y);
}
```

Will the missing value raise an error? **Answer:** The code will not raise a compilation error because of a missing semi-colon after `return` in `function`. Instead the compiler will go on until it find the next semi-colon which is after `x = 1`.

So, the instruction becomes `return x = 1;` which tells the compiler to assign `x` to 1 then return `x`.

1.7. Missing `break` in Switches

A `switch` statement with a missing `break` instruction at one or more cases **will not** raise an error and the code will be compiled normally.

```
 int x = 1;
int y;
switch(x)
{
    case 1:
        y = 3;
    case 2:
        y = 4;
        break;
}
printf("%d", y);
```

1.8. `if` and `else` Precendence

If `else` statement exists, it will be associated to the nearest, above of course, `if` statement.

```
int x = -1;
int y = 1;
int z = 0;
if(x > 0)
    if(y > 0)
        z = 1;
else
    z = 2;
printf("%d", z);
```

The output will be 0. Why?

Answer: Well, there is no change happened to `z` as the first `if` statement was false so the innner `if` statement didn't execute. And the `else` statement is associated to the nearest `if` statement which the is the second (inner) `if`.

Thus, this `else` statement is inside the scope of the outer `if` statment which won't be executed.

1.9. Syntactic Traps Prevention

- Don't rely on operation precenedence.
- Use brackets and parantheses.
- Never nest `if` statement with another `if` statement directly, use curly braces to define scope.
- Don't use empty semi-colons. Instead, add curly braces to open a scope and write a comment telling why you want nothing to execute.
- Avoid using postfix and prefix increment and decrement as possible (exeception in the case of the increment/decrement instruction in the `for` loop).

2. Semantic (Logical) Traps

2.1. Confusion Between Arrays and Pointers

Although arrays and pointers have similar syntax, arrays don't store addresses (unless the array was array of pointers) while pointers do.

Accessing the value of the array at a location using pointer notation is accepted, but it's preferable to use the square brackets notation for arrays and pointers notation for pointers for **readability**.

For example if we have 2D array `data[3][4]` so to access the second element in the second element of `data` i.e., the element at second row and second column

- Using pointer notation `*(*(data + 1) + 1)`
- Using square brackets notation `data[1][1]`

2.2. Overflow and Modulo

Overflow occurs for signed integers if the variable tries to store a value outside its range, so it changes its value and sign.

```
signed char x = 200; // [-128, 127]
printf("%d", x);
```

This code will result in `-56` as 200 is greater than 127, so overflow occurs.

Modulo occurs for unsigned integers. When a variable tries to store a value outside its range, it returns to zero when reaching maximum value (modulus effect [value % 2^{bits}]). Check C data types range but remember that the size of some data types (e.g., `int`) is compiler-dependent.

```
unsigned char x = 256; // [0, 255]
printf("%d", x);
```

Because 256 is greated than 255 and `x` is `unsigned char`, `x` will store 256%2^8 which will result in an output of `0`.

If `x` was 257, the output will be `1` (257%2^8).

The overflow and modulo can't be detected by compiler in the compilation time.

```
unsigned short address, offset; //[0, 65535]
if((address + offset) < 512)
{
    eepRead(address + offset);
}
else
{
    error();
}
```

Here because `unsigned short` has a range of [0, 65535] (2 bytes), if for example `address` was 35535 and `offset` was 30000. Then, the summation of `address` and `offset` is greater than 512, but also greater than 65535 which is the max range of `unsigned short` which will cause a modulo and returns a 1 which is less than 512.

So, you need to make sure that is `address + offset` can for any reason exceed the range, if so, you need to store it in a new variable that is capable of storing such extrema. Or, change the data type of `address` and `offset`.

2.3. Return

If there's a function that is supposed to return something, but you didn't add any `return` instruction

```
int add(int x, int y)
{
    int z = x + y;
}

int main(void)
{
    int res = add(3, 4);
    printf("%d", res);
}
```

If you run the code above, you will get 7 which seems to be a correct answer. **But,** that wasn't because `add` returned `z = x + y` , when `z = x + y` was executed, the value of `x + y` is stored temporally in the **accumlator** and when calling a function that is supposed to return something but no `return` instruction was found in the function. So, the value in the accumlator was moved to `res` .

If the code was

```
int add(int *x, int *y)
{
    (*y) = (*x) * 2;
    // Now, x points to a value of 3 and y points to 6
    int z = (*x) + (*y); // z will be 9
}

int main(void)
{
    int a = 3;
    int b = 2;
    int res = add(&a, &b);
    printf("%d", res);
}
```

The output will be `9` , because the last value in the accumlator before moving to `res` was 9.

But if we switched the lines in the `add` function,

```
int add(int *x, int *y)
{
    int z = (*x) + (*y); // z will be 5
    (*y) = (*x) * 2;
    // Now, x points to a value of 3 and y points to 6

}

int main(void)
{
    int a = 3;
    int b = 2;
    int res = add(&a, &b);
    printf("%d", res);
}
```

The output will be a random number that was retrieved from the accumlator into `res`, `res` now may contain the address of `y` (`b`) for example.

2.4. Side Effects

It's having an additional effect while returning a value (having increment inside an assignment or comparing operator)

```
int i = 10;
if(i++ < 10) printf("%d", i);
```

The code above will not print `i` because in `if(i++ < 10)`, the value of `i` will be incremented first before comparing to 10. Which may not be the intention, may be the programmer thought that the prefix is done after the comparison.

Another problem with the optimization of the conditions in C,

```
if(a < 5 && func(5) < 7>) x = 3;
```

if a is found to be less than 5 and because it is a logical-and operator, C optimization will not execute `func(5)` and will not compare it to 7 because either way the overall condition will be false. **If the code depends that `func()` will always be called, code execution is wrong.**

2.5. Scopes

Relying on the concept of scopes to name multiple variables with different **nested** scopes with the same name will cause confusion in maintainence and readablility.

The above case is applied for nested scopes (scope inside a higher scope), but if 2 variables are in 2 totally different scopes (e.g., 2 separate functions), it's ok to name variables with the same name.

OK

```
int func1(void)
{
    int x = 3;
    return x + 5;
}

int func2(void)
{
    int x = 4;
    return x - 3;
}
```

NOT OK

```
int main(void)
{
    int x = 3;
    int y = 0;
    {
        int x = 10;
        x++;
    }
    if(x > 4) y++;
}
```

2.6. Enums

Enums are signed integers so, arithmetic operation in them are valid to the compiler, but not preferable.

```
enum colors = {RED, GREEN, BLUE};

int first_color = RED;
int next_color = first_color + 1;
```

Here, the code takes the `RED` color and want to go the next color as `RED` is 0 so the next color is 1. But we already know as we wrote the `colors` enum that `BLUE` is the next color and there are no reason to retrieve `BLUE` using arithmetic operation on different enum members.

Also to avoid confusion, MISRA states that when defining an enum, if we want to explicity initalize a value for a member of the enum, either we set a value for all members, or define the value of the first element.

OK

```
enum colors = {RED = 1, GREEN = 2, BLUE = 2}
```

2.7. Semantic Traps Prevention

- Use square-brackets notation for arrays not pointer notation.
- Consider worst-case scenario for possible overflow and modulo cases.
- Don't apply enumeration or its members to arithmetic operations.
- In nested scopes, don't name their variables with the same name.

3. Pre-Processors Traps

NOTE: Some of the following function-like macros are likable to cause a presentation trap, justify the use of it if you are sure you will escape that trap.

3.1. Spaces

When creation a function-like macro,

```
#define f (x) ((x) - 1)

int main(void){
    int y = f(5);
}
```

After preprocessig, the code will be

```
int main(void){
    int y = (x)(5);
}
```

which will result in compilation error. Why?

Answer: because there is space between `#define f (x)` in which `f` will be replaced with `(x)`. To solve this problem,

```
#define f(x) ((x) - 1) // take care of spaces

int main(void){
    int y = ((5) - 1);
}
```

3.2. Parantheses

In function-like macros, all macro arguments must be surrounded by parantheses to avoid any unusual behavoir regarding order of execution

```
#define f(x, y) x * y

int main(void)
{
    int x = 3;
    int y = 4;
    printf("%d", f(x + 1, y));
}
```

The will result be `7`. Expected `16` ?

Answer: Because the code after preprocessing will be

```
int main(void)
{
    int x = 3;
    int y = 4;
    printf("%d", x + 1 * y);
}
```

And becuase of PEMDAS (operations precendence), `1 * y` will be performed first and then added to `x` resulting in `7`. To solve this trap,

```
#define f(x, y) (x) * (y)

int main(void)
{
    int x = 3;
    int y = 4;
    printf("%d", f(x + 1, y));
}
```

will be preprocessed into

```
int main(void)
{
    int x = 3;
    int y = 4;
    printf("%d", (x+ 1) * (y));
}
```

3.3. Implying Multiple Argument Evaluation

```
#define cube(x) ((x) * (x) * (x))

int getValue(void)
{
    static int i = 0;
    i++;
    return i;
}

int main(void)
{
    int y = 0;
    y = cube(getValue());
    printf("%d", y);
}
```

Will `getValue()` be applied once then passed to `cube` macro or vice versa ?

Answer: Preprcoessing will occur first so `getValue` will be called 3 times not just one. The code will be:

```
int main(void)
{
    int y = 0;
    y = (getValue()) * (getValue()) * (getValue());
    printf("%d", y);
}
```

So because of the `static` keyword in the function, `i` will be initialized once and will keep the last value for the next call so the ouptut will be `1 * 2 * 3` = `6`.

3.4. Creating New Aliases for Data Types

Macros shouldn't be used to create an alias.

```
#define PTR_CHAR char *

PTR_CHAR x, y;
```

What is the type of `x` and `y` ?

- One would assume `x` and `y` both of the type `PTR_CHAR` which is `char *`.
- However, the macro instructs the compiler to replace `PTR_CHAR` in the code with `char *`. When the text-replacement is done, this is the new code

```
char * x, y;
```

As the asterisk, when used to declare a pointer, is bound to the variable `x` not the data type `char`. So, the above code can be written as

```
char *x;
char y;
```

Now, this is much familiar presentation that * **`x` is a `char *`** * **`y` is a `char`**

A solution to avoid such problem is using `typedef` instead of a `#define` to give an alternative presentation (alias) to a data type.

```
typedef char * PTR_CHAR

PTR_CHAR x, y;
```

can be written as

```
char *x;
char *y;
```

Now, `x` and `y` are `char *`.

3.5. Multi-Statement Function-Like Macros

Macro functions that generate several statments that are not returning a value must be encapsulated with ```do{} while(0) `where every line is followed with a backslack` before going to the next line.

```
#define CHG_VALUE(VALUE, MASK, VAR) VAR &= ~MASK; VAR |= (MASK & VALUE);



int main(void)
{
    unsigned char x = 0xa0;
    CHG_VALUE(3, 0x03, x);
    printf("%d", x);
}
```

will be processed into

```
int main(void)
{
    unsigned char x = 0xa0;
    x &= ~0x03; x |= (0x03 & 3);;
    printf("%x", x);
}
```

Same if we wrote the macro,

```
#define CHG_VALUE(VALUE, MASK, VAR) do\
                                    {\
                                        VAR &= ~MASK;\
                                        VAR |= (MASK & VALUE);\
                                    } while(0);
```

Much better in readability.

Question: Why don't we just use `inline` keyword for function replacement ?

Answer: Because `inline` doesn't have the same configurations for all compilers if it was compatiable for that compiler in the first place.

3.6. Constant Macros

Macros defining constants must surround the constants with parantheses.

```
#define MAX 10 + 2

int main(void)
{
    printf("%d", MAX * 2);
}
```

The output will be `24`, right ?

Answer: Wrong, the replacement will make the code look like this,

```
int main(void)
{
    printf("%d", 10 + 2 * 2);
}
```

So `2 * 2` will be executed first then will be added to `10` to be `14`.

Using parantheses,

```
#define MAX (10 + 2)

int main(void)
{
    printf("%d", MAX * 2);
}
```

Preprpcessed code will be,

```
int main(void)
{
    printf("%d", (10 + 2) * 2);
}
```

3.7. Pre-Processors Traps Prevention

- All macros arguments, constants must be surrounded by parantheses.
- Avoid using macros for alias type definition, use `typedef` instead.
- Take care of possible evaluation of function or operator (prefix increment).

4. Conversions

When operations deal with different data types, conversion occur to make the data types the same.

- Operations for equal types don't need conversion.

```
// perfect case:

unsigned char x, y, z;
z = (unsigned char) 5;
y = (unsigned char) 6;
x = z + y; // all uchar (perfect case)
```

- If types are different, the samaller type is converted to the larger type

```
unsigned char x, z;
int y;
z = (unsigned char) 5;
y  10;
x = z + y; // z will be converted to int
```

For example,

- `long double` operations with smaller types require converting the samller types to `long double`.

```
long double x;
float y;
long double z;
x = 0.92;
y = 0.2;
z = x + y; // y will be converted to long double.
```

4.1. Integer Promotion

When arithmetic operations are done on integers (`char` is 1-byte `int`), including the uniary operators [+ ~ -]- integer promotion must be applied first.

It's promoted to a `signed int` or `int` if it can store the value (up to 31 bits), if it can't, the integer is promoted to `unsigned int` (up to 32 bits).

```
 int main(void)
{
    char a = 30, b = 40, c = 10;
    char d = (a * b) / c;
    printf("%d", d);
    return 0;
}
```

The result will be `120`.

```
 int main(void)
{
    char a = 0xfb; // char can only store 7 bits
    unsigned char b = 0xfb; // uchar can store 8 bits.
    printf("a = %x", a);
    printf("\nb = %x", b);
    if(a == b)
    {
        printf("\nSame");
    }
    else
    {
        printf("\nNot Same");
    }
    return 0;
}
```

The result will be

```
 a = fffffffb
b = fb
Not Same
```

Because:, when storing `0xfb` in a. It is promoted to a `signed int`. And to keep the sign bit *MSB*, it is padded by one, so it was printed to `fffffffb`.

When comparing `a` to `b`, `b` which was `unsigned char` was converted to `signed int`. But `b` was unsigned so it was positive. So when `b` is stored, it is stored as `000000fb`. That's why `Not Same` was printed.

NOTES: * Conversion (Promotion) will not occur if the types and signs of the operands in a operation are the same.

- If types are different but the sign is the same, convert smaller to bigger type.
- If signs are different but types are the same, convert the unsigned to the suitable signed type. If larger type is needed, convert both types to that larger type (`int`)
- If signs are different and types are different:
  - If the larger type is signed, convert the smaller unsigned to the larger signed.
  - If the larger type is unsigned, convert the smaller signed to the larger unsigned.