Lecture 6 - Safe C Programming

1. What is safe programming?

C suffers from several issues: 1. **Unspecified/Undefined parts** in which each compiler may be designed with different approach to deal with. **Example**: the size of `int` is 4 bytes in some compilers and 2 bytes in other ones.

2. **Non-portable parts** which may be valid for a compiler but unvalid for other compilers of different architectures.
3. **Risky parts** that may be illogical but the compiler won't tell you that. **Example**: Acessing the 20th element of an array that was declared with a size of 10.

Coding C in a safe manner will make our code limited in a subset of the language which is called a **Coding Standard**. One of the most famous standards is **MISRA-C** which is used frequently in the automative field.

NOTE: This is a useful link to look for some MISRA-C rules with examples. link. **Thanks to Eslam**

Example:

```
#define ARR_SIZE 10

int x[ARR_SIZE] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

for(int i = 0; i < ARR_SIZE; i++)
{
    x[(unsigned char) i] = 3;
}
```

This code will work fine because the `i` will always be less that 10 which is fine with the max limit of the max size of `unsigned char`, right ?

- Not really, what if we wanted to change the `MAX_SIZE` in the macro to 500 instead of 10. `define ARR_SIZE 500`
- Now when indexing the array when `i` is more than 255 will result in a logic error and only the indexes of 0-255 will be changed, some of these indexes will be changed more than once.
- i.e., when `i` is 144 (0x90), the value of `x` at index 144 (0x90) will be changed to 3. When `i` is 400 (0x190) when is casted to `unsigned char`, the value of `x` at index 140 (0x90) will be changed again to 3.
- MISRA-C tells us that when indexing the array, it should be integer or casted to integer.

---

2. Presentation Traps

Potential problems can be splitted into: * **Badly structured code:** * Hard to understand. * May hide bugs. * Hard in maintainence. * Hard to port. * **Language structure traps:** 1. Presentation traps. 2. Lexical traps. 3. Syntactic traps. 4. Semantic traps.

Examples for presentation traps are:

2.1. Unjustified usage of macros.

Example:

```
#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 *`.

Example:

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

int x = 3;
int y = 4;

int z = MUL(x+1, y);
```

Expected result would be 16 because an increment of `x` will happen first the new value of `x` is multiplied by `y` to be 16. Right?

- Wrong, again the `#define` directives are text-replacement based instructions. So, the code above can be written as

```
int x = 3;
int y = 4;

int z = x + 1 * y;

// z = 3 + 1 * 4 = 7
```

2.2. Using C Default Behaviors

An example of a default behavior is the keyword `static` when used alone, it becomes by default `static int`.

```
static i, x, v[] = {4, 7};
// So, i  is int, x is int, v is int[]
```

should be written,

```
static int i;
static int x;
static int v[] = {4, 7};
```

2.3. Avoid Any Coding Syntax That Would Cause Confusion

It's not preferable to use the comma operator to write multiple declarations or instructions in the same line, this would lead to misunderstanding of the actual intention of the instructions and the order of the execution.

For example,

```
int x;
int i = 0;
int v[10];
x = 4, i++, v[1] = i;
```

What will be the value of the `v[1]` ? Will it be 1 because the `i++` is executed first? Or, will it be 0 because the `v[1]` ?

Why thinking about it in the first place? Just write it

```
x = 4;
i++;
v[1] = i;
```

Another problem with not using the curly braces to define the instrcutions to be executed for loops and conditionals. Same for using different indenations for different scopes.

For example,

```
if(stop == TRUE)
    flag = ON;
    interlock = ON;
if(interlock == ON)
    open_doors();
```

- The `interlock = ON;` instruction has the same indentation as `flag = ON;` which would tell us that these lines will be executed when the conditon `stop == TRUE` is true.
- But, not using the curly braces here will cause that only `flag = ON;` (the first instruction after the condtion) only will be executed in case the condition is true and `interlock = ON;` will be executed anyways regardless of the condition was true or false.

A quick fix is to use the curly braces and indentation to provide better presentation of the code.

```
interlock = OFF;
if(stop == TRUE)
{
    flag = ON;
    interlock = ON;
}
if(interlock == ON)
{
    open_doors();
}
```

2.2 Presentation Traps Prevention

So to sum up, * Avoid creating data types aliases using `#define` . Use `typedef` instead.

- Avoid creating function-like macros. Like, `SET_BIT` , `CLEAR_BIT` , `TOGGLE_BIT` . Unless you justify using such kind of macros.
- Take care of the indenations, use code beautifier to help you.
- Use clear, meaningful, and standardized names.
- Don't depend on implicit or default behaviors of the C language.
- Don't declare multiple variables at the same statement. Define each separately.
- Avoid using comma operator.

---

3. Lexical Traps

3.1. Confusion Between the Syntax of Similar Operators

problems of using instructions and operators that may be used in different situations. For example,

- `=` and `==`
- `&` and `&&`
- `|` and `||`
- `!` and `!=`

For example,

```
int x = -1;
if(x = 1)
{
    printf("x is 1");
}
```

This code will always print `x is 1` although `x` was initalized to -1.

Even if your target was to assign `x` to 1 inside the `if` statement, I'd assume you also love summer more than winter. There's no reason to do such thing. So, write it

```
int x = -1;
if(x == 1)
{
    printf("x is 1");
}
```

The same for

```
x == y // Unuseful code....
```

Another lexical trap is between the bit-wise operations (& |) and logical operations (&& ||)

DON'T

```
if(x | y)
{
    // some instructions...
}
```

DO

```
if(x || y)
{
    // some instructions...
}
```

Another example is the difference between assignment operators

```
int x = 3;
x += 1; // x = 4
x =+ 1; // x = 1
```

3.2. Tokens

When the compiler executes a complex instruction, it divides it into different segments each segment is called **token**.

For example, * `x--y` means what ?

`x--` will decrement `x` then subtracts `y` from it.

write it

```
x--;
x = x - y; // or z = x - y;
```

- `y = x / *p; /* comment */` means what? Is `/* p` a part of the comment ?

`/* p` is part of the comment `/* p;` and the code will result in a compilation error because of the missing semi-colon ';' (it was part of the comment)

write it

```
z = *p;
y = x / z; // or y = x / (*p);
/* use block comment syntax only in its own line(s) not beside an instruction.
```

- `x-->y` means what ?

`x--` will decrement `x` first then will check if it is greater than `y`.

3.3 Constants Default Data Types

Another type of lexical traps is relying on C default types for constants

- For floating-point real constants, their default data type is `double`.
- For integer constants, their default data type is `signed int` or `int`.

An example for such trap,

```
float x = 0.7;

if(x == 0.7)
{
    printf("True\n");
}
else
{
    printf("False\n");
}
```

This code will result in `False`. Why?

0.7, when stored as `float`, is stored with precision error (not exactly 0.7 but 0.699999999). The limitations of the exponent and mantissa in IEEE standard is different from `float` and `double` which is the default data type for the constant 0.7.

A simple solution is to cast the constant 0.7 to `float` or assign `x` to be `double`.

```
double x = 0.7;

if(x == 0.7)
{
    printf("True\n");
}
else
{
    printf("False\n");
}
```

or

```
float x = 0.7;

if(x == (float)0.7)
{
    printf("True\n");
}
else
{
    printf("False\n");
}
```

But the first solution is better for its portablility.

3.4. Lexical Traps Prevention

- Write your code clearly, use clear spaces between operators.
- Limit the usage of assignment operators other than normal assignment (use `=` instead of `+=` or `-=`).
- Avoid depending on the default data types for constants.
- Avoid depending on C precedent (PEMDAS).
- Take into consideration of the possible confusion between operators.