

**AIN SHAMS UNIVERSITY**

**FACULTY OF ENGINEERING**



## **CSE211: Introduction to Embedded Systems**

### **Lab2**

**Name: Salma Sherif Mohamed**

**ID:20P5762**

## Task1:

```
Main.c Lab2.h + 3zrra23bk NEW C RUN
```

```
1 #include "Lab2.h"
2 int main () {
3     int8_t my_int8 = -10;
4     uint8_t my_uint8 = 20;
5     int16_t my_int16 = -30;
6     uint16_t my_uint16 = 40;
7     int32_t my_int32 = 50;
8     uint32_t my_uint32 = 60;
9     int64_t my_int64 = 70;
10    uint64_t my_uint64 = 80;
11    float32 my_float = 2.5;
12    double64 my_double = 2.4;
13
14    int8_ptr my_int8_ptr = &my_int8;
15    uint8_ptr my_uint8_ptr = &my_uint8;
16    int16_ptr my_int16_ptr = &my_int16;
17    uint16_ptr my_uint16_ptr = &my_uint16;
18    int32_ptr my_int32_ptr = &my_int32;
19    uint32_ptr my_uint32_ptr = &my_uint32;
20    int64_ptr my_int64_ptr = &my_int64;
21    uint64_ptr my_uint64_ptr = &my_uint64;
22    float32_ptr my_float_ptr = &my_float;
23    double64_ptr my_double_ptr = &my_double;
24
25    printf ("%d\n",my_int8);
26    printf ("%d\n",my_uint8);
27    printf ("%d\n",my_int64);
28    printf ("%f\n",my_float);
```

STDIN

Input for the program ( Optional )

---

Output:

```
-10
20
70
2.500000
2.400000
```

```
Main.c Lab2.h + 3zrra23bk NEW C RUN
```

```
1 typedef signed char int8_t;
2 typedef unsigned char uint8_t;
3 typedef signed short int16_t;
4 typedef unsigned short uint16_t;
5 typedef signed long int32_t;
6 typedef unsigned long uint32_t;
7 typedef signed long long int64_t;
8 typedef unsigned long long uint64_t;
9 typedef float float32;
10 typedef double double64;
11
12
13 typedef int8_t* int8_ptr;
14 typedef uint8_t* uint8_ptr;
15 typedef int16_t* int16_ptr;
16 typedef uint16_t* uint16_ptr;
17 typedef int32_t* int32_ptr;
18 typedef uint32_t* uint32_ptr;
19 typedef int64_t* int64_ptr;
20 typedef uint64_t* uint64_ptr;
21 typedef float* float32_ptr;
22 typedef double* double64_ptr;
```

STDIN

Input for the program ( Optional )

---

Output:

```
-10
20
70
2.500000
2.400000
```

### Difference between typedef and #define for type definition:

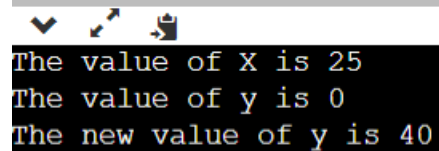
The main difference between typedef and #define for type definition is that typedef is a keyword that is processed by the compiler, while #define is a preprocessor directive that is

processed by the preprocessor. This means that typedef definitions have more scope and can be used in more places than #define definitions.

Another difference is that typedef definitions can be used to create new type names for existing data types, while #define definitions can only be used to create aliases for existing data types. This means that typedef definitions can be used to create more complex and abstract data types, such as structures and unions.

## Task2:

```
1  #include <stdio.h>
2
3  #define X 25
4
5  int y;
6
7  int main () {
8      printf("The value of X is %d \n" ,X);
9      printf("The value of y is %d \n" ,y);
10     y=40;
11     printf("The new value of y is %d \n" ,y);
12     return 0;
13 }
```



```
✓ ↩ 📄
The value of X is 25
The value of y is 0
The new value of y is 40
```

When are they resolved in the build process?	X is resolved during the preprocessing phase of the build process.	y is resolved during the compilation phase of the build process.
Where is each one of them stored?	X is stored in the symbol table.	y is stored in the data segment of the program.
How can each of them be used in the code?	X can be used as a constant value in the code.	y can be used as a variable in the code.
Which of them could be altered and why?	X cannot be altered because it is a constant.	y can be altered because it is a variable.

## Lab B

### Task1:

**1: • Create a macro that adds two parameters. Also Create a function that adds two variables and returns the sum.**

**Then Compare the differences between the macro and the function in the following points:**

○ **When will each be resolved in the build process?**

- A macro is resolved during the preprocessing stage, which is before the compilation. The preprocessor replaces every occurrence of the macro name with its definition in the source code.

- A function is resolved during the compilation or linking stage, depending on whether it is defined in the same source file or in a different one. The compiler generates code for each function call and replaces it with an address or a label.

○ **What will happen when you pass wrong parameters to each one of them?**

- A macro does not perform any type checking or error checking on its parameters. It simply performs textual substitution.

This can lead to unexpected results or errors if the parameters are not compatible with the expression.

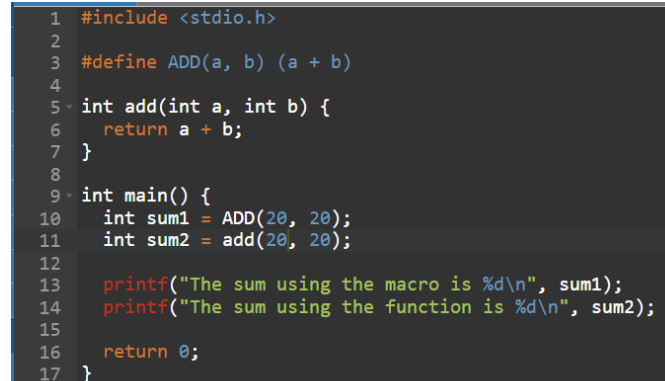
- A function performs type checking and error checking on its parameters. It will not accept parameters that are not compatible with its declaration.

If you pass a wrong parameter to a function, you will get a compile-time error or a warning.

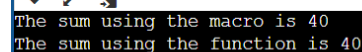
○ **What will be the sequence in the function call and how will that differ in the macro like function?**

- A function call follows a specific sequence of steps to pass the parameters, execute the function body, and return the value.

For example, when you call `add(3, 4)` , the following steps happen:



```
1 #include <stdio.h>
2
3 #define ADD(a, b) (a + b)
4
5 int add(int a, int b) {
6     return a + b;
7 }
8
9 int main() {
10     int sum1 = ADD(20, 20);
11     int sum2 = add(20, 20);
12
13     printf("The sum using the macro is %d\n", sum1);
14     printf("The sum using the function is %d\n", sum2);
15
16     return 0;
17 }
```



```
The sum using the macro is 40
The sum using the function is 40
```

- \* The values of 3 and 4 are pushed onto the stack or stored in registers.
- \* The program counter jumps to the address or label of the add function.
- \* The function body executes and calculates the sum of x and y.
- \* The return value is stored in a register or popped from the stack.
- \* The program counter returns to the caller and resumes execution.

- A macro like function does not follow any sequence of steps. It simply expands into its definition wherever it is used. For example, when you use `ADD(3, 4)`, it expands into `((3) + (4))`, which is then evaluated by the compiler as part of the expression.

● **What will happen if the variable is declared in the .h file? Then the .h file is included in 2 .c files? Why do we add the variable extern declaration in the .h file?**

- If a variable is declared in a .h file without using the ``extern`` keyword, it will be defined as a global variable in every .c file that includes it. This can cause multiple definition errors or conflicts if different .c files try to access or modify it. For example, if you declare `int x = 10;` in a .h file and include it in two .c files, you will have two copies of x with different values.

- To avoid multiple definition errors or conflicts, we add the variable extern declaration in the .h file. This tells the compiler that the variable is defined somewhere else and only declares its name and type. Then we define it only once in one of the .c files. For example, if we declare `extern int x;` in a .h file and include it in two .c files, we will have only one copy of x that can be accessed by both files

## Lab B:

### Task 2:

#### Syntax errors:

- Missing parenthesis
- Missing semicolon
- Mismatched curly braces
- Undeclared variable
- Invalid function call

```
1 |  
2 | #include <stdio.h>  
3 |  
4 | int main() {  
5 |     if (x > 10) {  
6 |         printf("x is greater than 10");  
7 |     }  
8 | }
```



input

Compilation failed due to following error(s).

```
main.c: In function 'main':  
main.c:5:7: error: 'x' undeclared (first use in this function)  
5 |     if (x > 10) {  
  |         ^
```

main.c:5:7: note: each undeclared identifier is reported only once for each function

## Semantic errors:

- Trying to add two strings together
- Trying to divide by zero
- Accessing an array element out of bounds
- Using a pointer to an invalid memory location
- Calling a function that has not been defined

```
1
2 #include <stdio.h>
3
4 int main() {
5     char *str1 = "Hello";
6     char *str2 = "World";
7
8     int sum = str1 + str2; // Semantic error
9
10    return 0;
11 }
```

input

Compilation failed due to following error(s).

```
main.c: In function 'main':
main.c:8:18: error: invalid operands to binary + (have 'char *' and 'char *')
8 |     int sum = str1 + str2; // Semantic error
  |                  ^
```

## Run-time errors:

- Division by zero
- Array index out of bounds
- String index out of bounds
- Null pointer dereference
- Segmentation fault

```
1
2 #include <stdio.h>
3
4 int main() {
5     int arr[10];
6
7     // Access an array element out of bounds
8     printf("%d\n", arr[10]);
9
10    return 0;
11 }
```

-2019826944



## Linker errors:

- Missing function definition
- Missing library file
- Duplicate symbol

```
1
2 #include <stdio.h>
3
4 void my_function();
5
6 int main() {
7     // Call my_function()
8     my_function();
9
10    return 0;
11 }
```

input

Compilation failed due to following error(s).

```
/usr/bin/ld: /tmp/cce2ldTe.o: in function `main':
main.c:(.text+0xe): undefined reference to `my_function'
collect2: error: ld returned 1 exit status
```

## Logical errors:

- Infinite loop
- Off-by-one error
- Using the wrong operator
- Missing a break statement
- Using an uninitialized variable

```
main.c
1
2 #include <stdio.h>
3
4 int main() {
5     int i = 0;
6     while (i < 10) {
7         printf("%d\n", i);
8     }
9
10    return 0;
11 }
```

