

2022-09-13 LECTURE 09 – MODULAR PROGRAMMING – II

- contents
 - function call
 - concepts discussed in the lecture
 - exercises
- function call
 - a function can call as many functions as required
 - when a function calls another function following memory structures are used
 - **function call stack**: stores order of all function calls
 - **function stack frame**: stores information of a single function call
 - function call stack or system stack
 - a **data structure** which holds function call status, used to preserve the order of flow control during function calls
 - **system stack** is used to store information about a function calls (where it gets the name "function call stack")
 - this stack stores parameters for the function and a **return address** where the program should pick up when the function is finished or return
 - function stack frame
 - data set that should be placed at **function call stack** when a function is called
 - it contains caller function's returning address
 - address of the next instruction after the function call statement in the caller function
 - where the program control would be transferred after the completion of the **called function**
 - parameters sent by the **caller function**
 - **local variables** of the called function
 - when any function (**caller**) calls another function (**called**), then **function stack frame** of called is placed at the opt of **function stack frame** of the function caller at **function call stack** in the main memory
 - **stake overflow**
 - when the memory of the stack is unavailable to hold any further function stack frame, program crashes and a **system error** called "stack overflow" is printed at the console
 - **example (demonstration of a user defined function)**: a program that call a function named **square**, which takes an integer as parameter, compute its square and returns the value of the square to the caller function (the main), and then the program prints the square at the console.
// L09-C01
 1. `#include <stdio.h>`
 - 2.
 3. `int square (int);`
 4. `int square_of_square (int);`

```

5. int square_of_square_of_square (int);
6.
7. int main (void) {
8.     int n;
9.     n = square (10);
10.    printf("%d\n", n);
11.    n = square_of_square (10);
12.    printf("%d\n", n);
13.    n = square_of_square_of_square (10);
14.    printf("%d\n", n);
15. }
16.
17. int square (int x) {
18.     int a;
19.     a = x * x;
20.     return a;
21. }
22. int square_of_square (int x) {
23.     int a;
24.     a = square(x) * square (x);
25.     return a;
26. }
27. int square_of_square_of_square (int x) {
28.     int a;
29.     a = square_of_square (x) * square_of_square (x);
30.     return a;
31. }
▪ output: 100
▪ output: 10000
▪ output: 100000000

```

- function square: a function that takes an integer as argument and returns its square

- prototype: `int square (int);`
- definition:

```
int square (int i) {
    int x;
    x = i * i;
    return x;
}
```

- function call: `x = square (10);`

- function stack:

- **operating system** calls **main** function, and **function stack frame** of the main function is placed at **function call stack**, then when the **main function** calls **square function**, a function stack frame of the **square**

function pushed at the top of **function call stack**, hence the **function call stack** looks as follows

- square
- main
- when the **square function** completes its execution and returns a value 100 back to the **main function**, the **function stake frame** of the **square function** is removed from the **function call stack** and the control is transferred back to the **main function** hence the **function call stack** looks as follows
 - main
- function square_of_square: a function that takes an integer as argument and returns its square of its square
 - prototype: `int square_of_square (int);`
 - definition:

```
int square_of_square (int i) {
    int x;
    x = square (i) * square (i);
    return x;
}
```
 - function call: `x = square_of_square (10);`
 - function stack:
 - **operating system** calls **main** function, and **function stack frame** of the **main** function is placed at **function call stack**, then when the **main function** calls **square_of_square function**, a function stack frame of the **square_of_square function** is pushed at the top of **function call stack**, hence the **function call stack** looks as follows
 - square_square
 - main
 - then when the **square_of_square function** calls first **square function**, a function stack frame of the **square function** is pushed at the top of **function call stack**, hence the **function call stack** looks as follows
 - square
 - square_square
 - main
 - when the **square function** completes its execution and returns a value 100 back to the **square_of_square function**, the **function stake frame** of the **square function** is removed from the **function call stack** and the control is transferred back to the **square_of_square function** hence the **function call stack** looks as follows
 - square_of_square
 - main
 - when the **square_of_square function** calls **second square function**, a function stack frame of the **square function** is pushed at the top of **function call stack**, hence the **function call stack** looks as follows
 - square

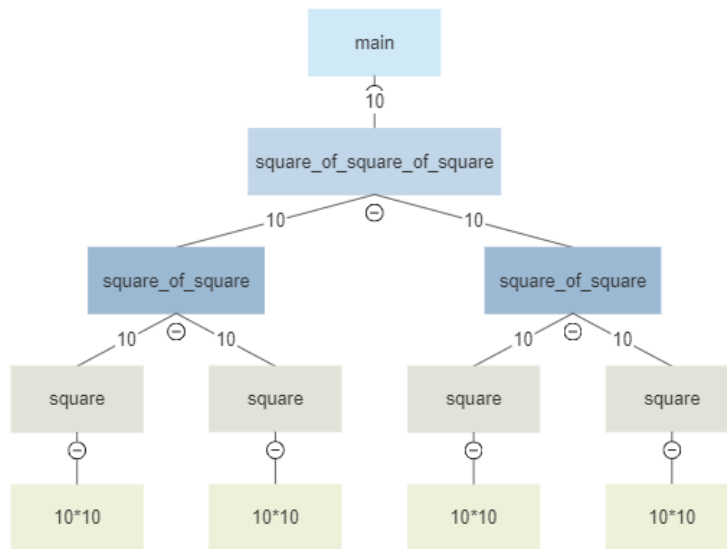
- square_of_square
 - main
- when the **square function** completes its execution and returns a value back to the **square_of_square function**, the **function stake frame** of the **square function** is removed from the **function call stack** and the control is transferred back to the **square_of_square function** hence the **function call stack** looks as follows
 - square_of_square
 - main
- when the **square_of_square function** completes its execution and returns a value 10,000 back to the **main function**, the **function stake frame** of the **square_of_square function** is removed from the **function call stack** and the control is transferred back to the **main function** hence the **function call stack** looks as follows
 - main
- function square_of_square_of_square: a function that takes an integer as argument and returns its square of square of square
 - prototype: int square_of_square_of_square (int);
 - definition: int square_of_square_of_square (int i) {
 - int x;
 - x = square_of_square (i) * square_of_square (i);
 - return x;
 - }
 - function call: x = square_of_square_of_square (10);
 - function stake:
 - **operating system** calls **main** function, and **function stack frame** of the **main function** is placed at **function call stack**, then when the **main function** calls **square_of_square_of_square function**, a function stack frame of the **square_of_square_of_square function** is pushed at the top of **function call stack**, hence the **function call stack** looks as follows
 - square_of_square_of_square
 - main
 - **square_of_square_of_square function** calls first **square_of_square function**, and **function stack frame** of the **square_of_square function** is pushed at **function call stack**, then when the **square_of_square function** calls its **first square function**, a function stack frame of the **square function** is pushed at the top of **function call stack**, hence the **function call stack** looks as follows
 - square
 - square_of_square
 - square_of_square_of_square
 - main
 - when the first **square function** completes its execution and returns a value 100 back to the **square_of_square function**, the **function stake**

frame of the **square function** is removed from the **function call stack** and the control is transferred back to the **square_of_square function** hence the **function call stack** looks as follows

- square_of_square
- square_of_square_of_square
- main
- then when the **square_of_square function** calls second **square function**, a function stack frame of the **square function** is pushed at the top of **function call stack**, hence the **function call stack** looks as follows
 - square
 - square_of_square
 - square_of_square_of_square
 - main
- when the **square function** completes its execution and returns a value 10,000 back to the **square_of_square function**, the **function stake frame** of the **square function** is removed from the **function call stack** and the control is transferred back to the **square_of_square function** hence the **function call stack** looks as follows
 - square_of_square
 - square_of_square_of_square
 - main
- when the **square_of_square function** completes its execution and returns a value 10,000 back to the **square_of_square_of_square function**, the **function stake frame** of the **square_of_square function** is removed from the **function call stack** and the control is transferred back to the **square_of_square_of_square function** hence the **function call stack** looks as follows
 - square_of_square_of_square
 - main
- **square_of_square_of_square function** calls second **square_of_square function**, and **function stack frame** of the **square_of_square function** is pushed at **function call stack**, then when the **square_of_square function** calls its **first square function**, a function stack frame of the **square function** is pushed at the top of **function call stack**, hence the **function call stack** looks as follows
 - square
 - square_of_square
 - square_of_square_of_square
 - main
- when the first **square function** completes its execution and returns a value 100 back to the **square_of_square function**, the **function stake frame** of the **square function** is removed from the **function call stack** and the control is transferred back to the **square_of_square function** hence the **function call stack** looks as follows

- square_of_square
 - square_of_square_of_square
 - main
- when the **square_of_square function** calls second **square function**, a function stack frame of the **square function** is pushed at the top of **function call stack**, hence the **function call stack** looks as follows
 - square
 - square_of_square
 - square_of_square_of_square
 - main
- when the **square function** completes its execution and returns a value 10,000 back to the **square_of_square function**, the **function stake frame** of the **square function** is removed from the **function call stack** and the control is transferred back to the **square_of_square function** hence the **function call stack** looks as follows
 - square_of_square
 - square_of_square_of_square
 - main
- when the second **square_of_square function** completes its execution and returns a value 10,000 back to the **square_of_square_of_square function**, the **function stake frame** of the **square_of_square function** is removed from the **function call stack** and the control is transferred back to the **square_of_square_of_square function** hence the **function call stack** looks as follows
 - square_of_square_of_square
 - main
- when the **square_of_square_square function** completes its execution and returns a value 100,000,000 back to the **main function**, the **function stake frame** of the **square_of_square_square function** is removed from the **function call stack** and the control is transferred back to the **main function** hence the **function call stack** looks as follows
 - main
- **function call tree**
 - is a diagram which depicts the sequence (or order) of function calls
 - following tree shows the function call tree from the function call from the line 13 of the program **L09-C01**
 - **main function**
 - call **square_of_square_of_square**
 - calls left **square_of_square**
 - calls left **square**
 - returns (10*10)
 - call right **square**
 - returns (10*10)
 - returns ((10*10) * (10*10))

- calls right **square_of_square**
 - calls left **square**
 - returns $(10*10)$
 - call right **square**
 - returns $(10*10)$
 - returns $((10*10) * (10*10))$
- returns $((((10*10) * (10*10)) * ((10*10) * (10*10))))$
- main receives $((((10*10) * (10*10)) * ((10*10) * (10*10))))$



- concepts discussed in the lecture
 - function call stack or system stack, function stack frame, return address, called function, caller function, local variables, stack overflow, system error, function call tree
- exercises