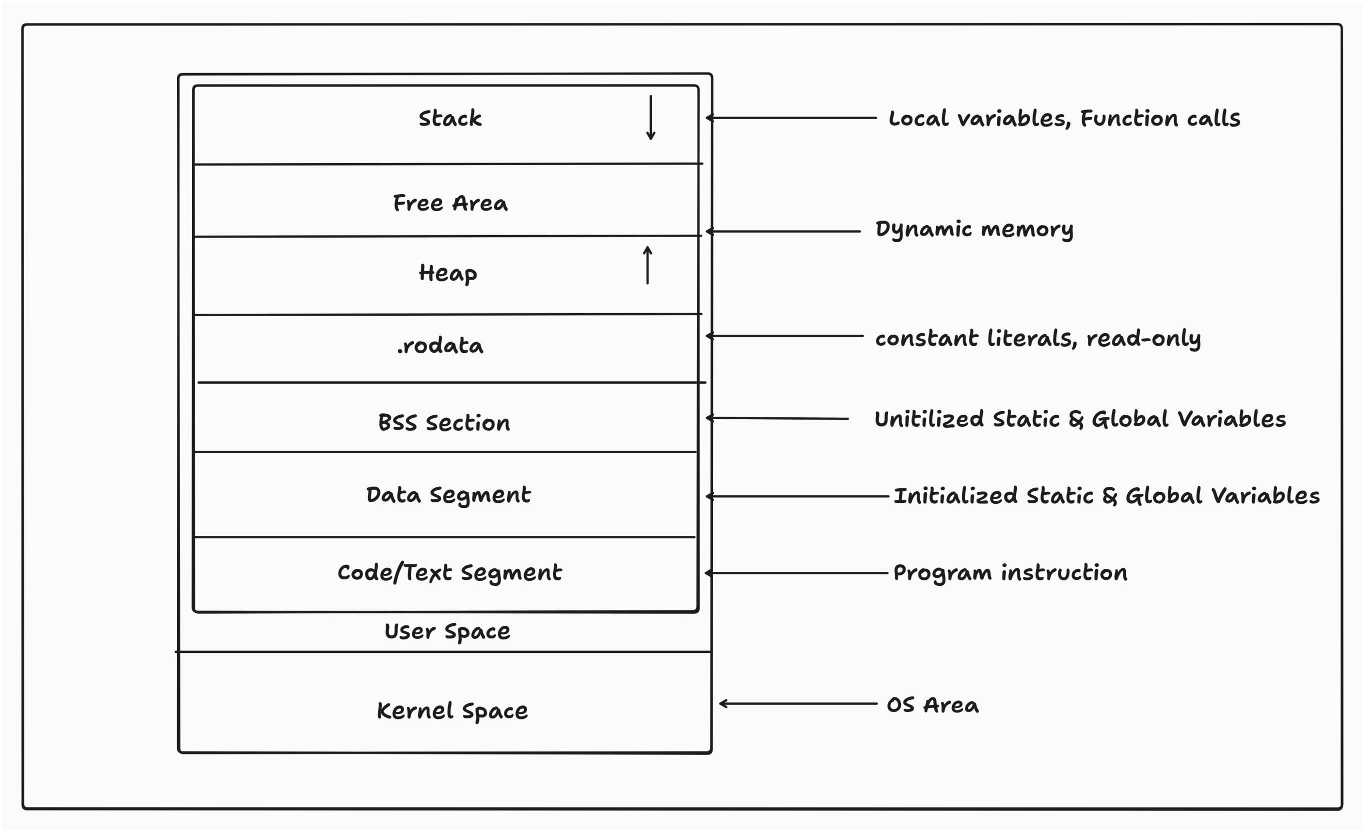
24/03/26 DAY 2

Memory Layout

Memory is divided into segments as seen in the below image. Each segment has its own purpose and it is used to store different types of data.


Stack
Free Area
Heap
.rodata
BSS Section
Data Segment
Code/Text Segment
User Space
Kernel Space
Local variables, Function calls
Dynamic memory
constant literals, read-only
Unitilized Static & Global Variables
Initialized Static & Global Variables
Program instruction
OS Area

Scope & lifetime of variables

Scope:

- It is something that is related where it will be accessible.
- A variable can be accessed within the scope it is declared.
- It can be global, local or block scope.

Lifetime:

- How long it will stay in memory.
- Global variables will stay in memory until the program ends
- Local variables will stay in memory until the function ends
- Block scope variables will stay in memory until the block ends.

```
#include <stdio.h>

int num = 100;  // global variable
int main() {

    printf("num: %d \n", num);

    int num = 10;    // local variable

    printf("num: %d \n", num);

    {
        printf("num: %d \n", num);
        int num = 20;    // block scope local variable
        printf("num: %d \n", num);
    } // scope ends here

    printf("num: %d \n", num);

    return 0;
}
```

Function Introduction

A block of code which perform a task for which it is designed for. Reusalability.

```
int increment(int num, int incr){
    num = num + incr;
    return num;
}

int main() {
    int num = 10;    //initilization

    int result = increment(num, 2);

    printf("num: %d", result); //print

    return 0;
}
```

Function prototype/definition/call

- **Function Prototype**: It is a declaration of a function which tells the compiler about the function name, return type and parameters. It is also called as function declaration.
- **Function Definition**: It is the actual implementation of the function which contains the business logic. It is also called as function body.
- **Function Call**: It is the process of invoking a function to perform a task. It is also called as function invocation.

```
int add(int,int);   // function declaration/prototype

int main(){
    int a = 10;
    int b = 20;

    int result = add(a,b);  // function call

    printf("%d",result);

    return 0;
}

int add(int a, int b){  // function definition
    return a+b;
}
```

Function Design

```
return_type function_name(function_parameters...){
    // bussiness logic
    return_statement;
}
```

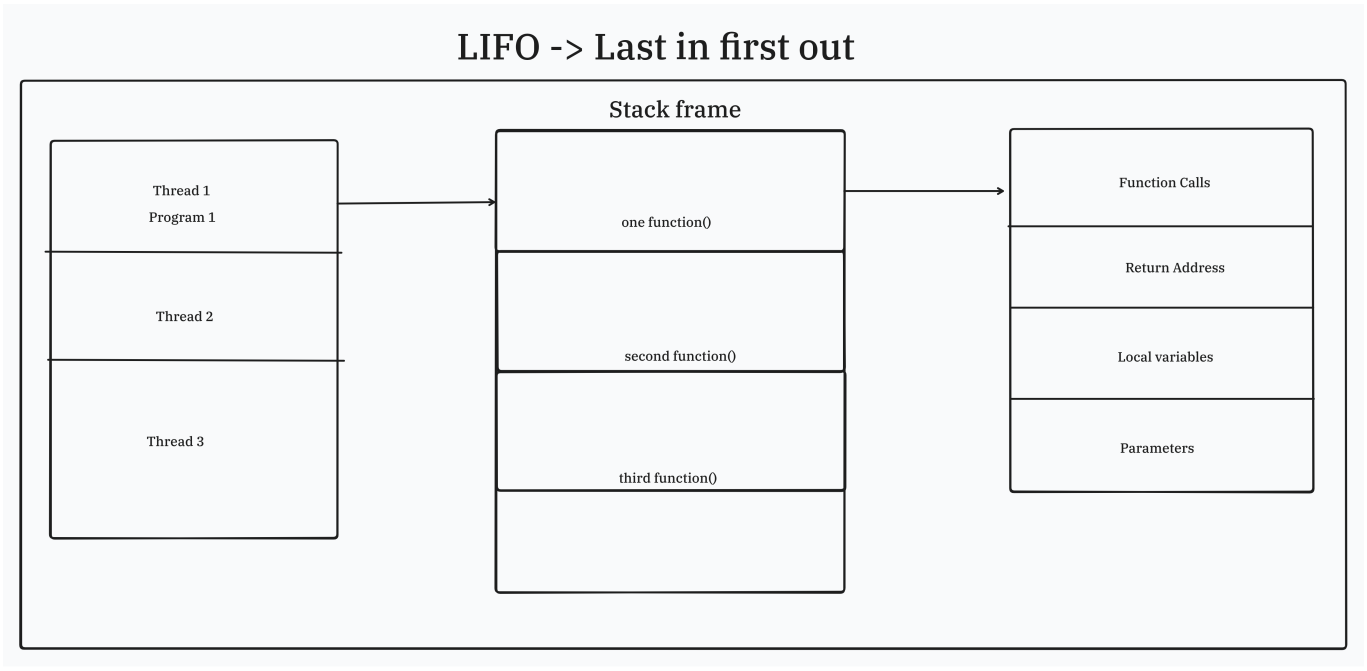
- Function Name must be meaningful - A function is doing some work it must tell what actually it is doing.
- Function should not affect the global values
- Function must do a single task
- Parameters/arguments must be limited(as per the requirement).

Activation record concept

- Activation Record is a block of memory which is created when a function is called. It is also called as stack frame.

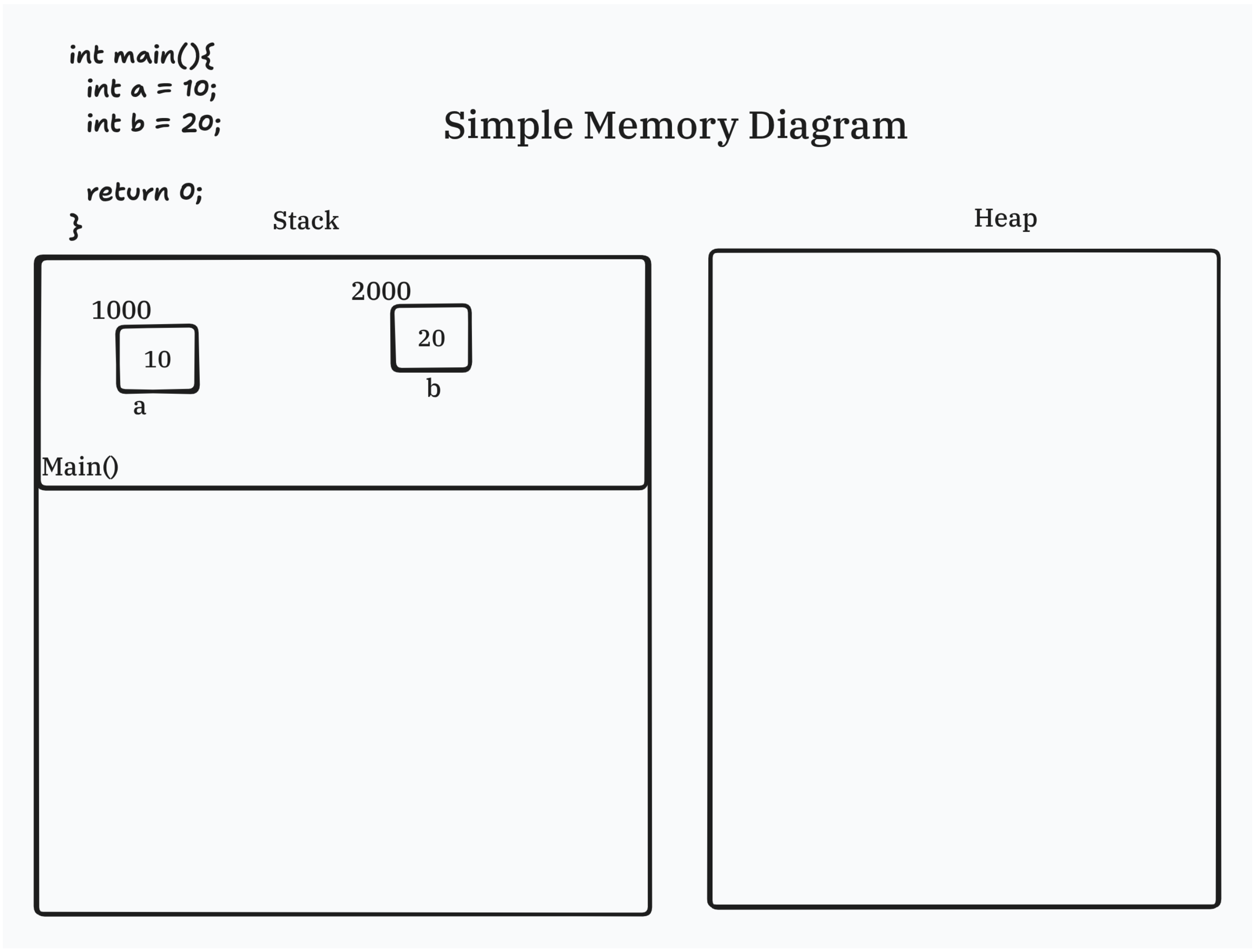
It consist of the following:

- Return address: It is the address of the instruction which will be executed after the function call
- Parameters: It is the list of parameters which are passed to the function
- Local variables: It is the list of local variables which are declared inside the function
- Function calls: It is the list of function calls which are made inside the function



Simple memory diagram

> We can draw a simple memory diagram to understand how the memory is allocated for the variables and functions. How the variables are stored and how there values are changed when we call the function.



Call by value & Call by reference

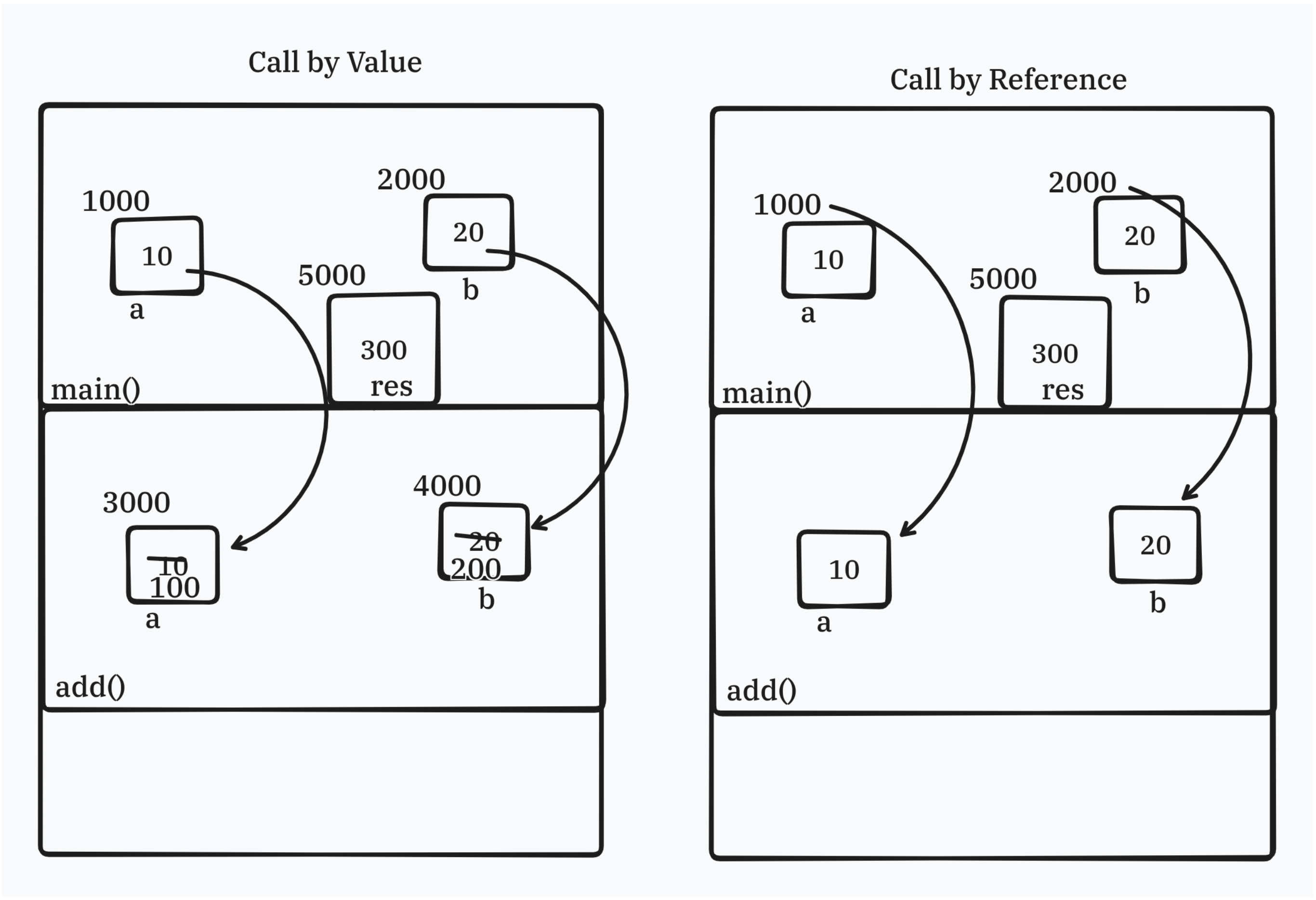
Call by value:

- In this method, the value of the variable is passed to the function.
- The function creates a copy of the variable and works with that copy.
- Any changes made to the variable inside the function will not affect the original variable outside the function.

Call by reference:

- In this method, the reference of the variable is passed to the function.
- The function works with the original variable and any changes made to the variable inside the function will affect the original variable outside the function.

```
#include <stdio.h>
int add(int a, int b){  // call by value
    a = 100;
    b = 200;

    printf("Inside add \n");
    printf("Address of a: %p \n", &a);
    printf("Address of b: %p \n", &b);

    return a+b;
}

int add(int &a, int &b){    // call by reference
    a = 100;
    b = 200;

    printf("Inside add \n");
    printf("Address of a: %p \n", &a);
    printf("Address of b: %p \n", &b);

    return a+b;
}

int main() {
    int a = 10;
    int b = 20;

    printf("a: %d \n", a);
    printf("b: %d \n", b);
    printf("Address of a: %p \n", &a);
    printf("Address of b: %p \n", &b);

    int res = add(a,b);

    printf("res: %d \n", res);

    printf("a: %d \n", a);
    printf("b: %d \n", b);

    return 0;
}
```



Reference variable

- Reference variable is a variable which is an alias for another variable.
- It is created using the & operator.
- It is used to create a reference to a variable.
- It is also called as reference type.
- Referece variable must be initialized at the time of declaration.
- Reference variable != Pointer variable

> Reference variable does not have its own memory location. It shares the same memory location as the variable it is referencing.

```
int a = 10;
int &ref = a;    // ref is a reference variable which is an alias for a

printf("a: %d \n", a);  // 10
printf("ref: %d \n", ref);  // 10

ref = 20;    // changing the value of ref will change the value of a
```

```
printf("a: %d \n", a);  // 20
printf("ref: %d \n", ref);  // 20


printf("Address of a: %p \n", &a);  // address of a
printf("Address of ref: %p \n", &ref);  // address of a
```

Address-of operator

& is the address-of operator

- It is used to get the address of a variable.
- It is also used to create a reference variable.
- It is also used to pass the variable by reference to a function.

```
int a = 10;
// &a will give the address of a
printf("Address of a: %p \n", &a);  // address of a

// creating a reference variable
int &ref = a;   // ref is a reference variable which is an alias for a
printf("Address of ref: %p \n", &ref);  // address of a
```

Pointer introduction

Pointer is a variable which stores the address of another variable.

- Name of pointer variable must be meaningful - A pointer variable is used to store the address of another variable, it must tell what actually it is storing.
- Pointer variable must be initialized at the time of declaration.
- Pointer without initialization will point to some garbage address and it can lead to undefined behavior if we try to access it.
- Pointer variable without initialization is called as wild pointer.

```
int a = 10; // normal variable
int *ptr = &a;  // pointer variable which stores the address of a
```

Pointer declaration & Pointer assignment

```
int a = 10; // normal variable
int *ptr;   // pointer declaration
ptr = &a;   // pointer assignment
```

```
printf("a: %d \n", a);  // 10
printf("ptr: %p \n", ptr);  // address of a
printf("*ptr: %d \n", *ptr);    // 10
```

Dereferencing operator

* is the dereferencing operator

- It is used to get the value stored at the address which is stored in the pointer variable
- It is also used to change the value stored at the address which is stored in the pointer variable.

```
int a = 10; // normal variable
int *ptr = &a;  // pointer variable which stores the address of a

// Dereferencing
*ptr = 20;  // changing the value stored at the address which is stored in
the pointer variable

printf("a: %d \n", a);  // 20
```

NULL vs nullptr

NULL is a macro which is defined in stddef.h header file. It is used to represent a null pointer. It is an integer constant with value 0.

```
#include <stdio.h>
#include <stddef.h>

void func(int val){
    printf("Inside int function \n");
}

void func(int *ptr){
    printf("Inside pointer function \n");
}

int main() {
    int *ptr = NULL;    // ptr is a null pointer

    func(ptr);  // it will call the pointer function

    func(NULL); // causes ambiguity

    func(nullptr); // it will call the pointer function

    return 0;
}
```