Module 7 – Part 1 - Functions

BITS Pilani
Pilani Campus

Dr. Jagat Sesh Challa
Department of Computer Science & Information Systems

Module Overview

- What are Functions?
- Function Declaration
- Function Definition
- Function Call
- Functions in Memory
- Call by Value
- Exercises

innovate achieve lead

BITS Pilani
Pilani Campus

Functions

Functions

We have already seen functions

`main()` function is the place where any C program starts its execution

```
/* myfirst.c: to compute the sum of two numbers */
#include<stdio.h> //Preprocessor directive
/*Program body*/
int main()
{
int a, b, sum;  //variable declarations
printf("Please enter the values of a and b:\n");
scanf("%d %d", &a, &b);
sum = a + b;  // the sum is computed
printf("The sum is %d\n", sum);
return 0; //terminates the program
}
```

What are functions?

- Functions are *"Self contained program segment that carries out some specific well-defined task"*

- A function
  - *processes information that is passed to it from the calling portion of the program*, and
  - *returns a single value*

- Every C program has one or more functions

Sum of two numbers:

```
int sum(int a, int b)
{
        int total;
        total = a + b;
        return total;
}
int main()
{
        int x,y,z;
        x = 5, y = 4;
        z = sum(x,y);
        printf("Sum is %d:",z};
}
```

Two values received by the "sum" function

"sum" function computes the sum of two values and returns it

"main" function calls "sum" function passing values of x and y and receives their sum.

Using Functions

Functions are

- Declared
- Defined
- Called

```
/* myProg.c */
#include <stdio.h>
int sum(int a, int b);
int sum(int a, int b)
{
        int total;
        total = a + b;
        return total;
}
int main()
{
        int x,y,z;
        x = 5, y = 4;
        z = sum(x,y);
        printf("Sum is %d:",z};
}
```

Function Declaration

Function Definition

Function Call

innovate achieve lead

BITS Pilani
Pilani Campus

Function Declaration

Function Declaration

- Gives information to the compiler that the function may later be used in the program
  - Declarations appear before definitions

- Although optional in many compilers, it is a good practice to use

- Syntax:

```
<return_type> <function_name> (list-of-typed-parameters);
```

- Example:
  - `float sqrt(float x);`
  - `void average(float, float);`

Function Declaration (Contd.)

```
int average(float a, float b);
```

Return type of the function

- Any valid data type in C
- `void` in case a function does not return anything explicitly

Name of the function

- Any valid identifier name

Optional list of parameters

- A comma-separated list of type-name pairs
  - Or just types
- It can be an empty list

Another example

```
void average(float, float);
```

Note that declarations doesn't mandate specifying the variable names

innovate achieve lead

BITS Pilani
Pilani Campus

Function Definition

Function Definition

Syntax:

```
return_type function_name (list_of_parameters)
{
    function body
}
```

Parts of function definition:

1. Type of the value it returns
2. Name of the function
3. Optional list of typed parameters
4. Function body

Function Definition

```
int average(float a, float b)
{
    int avg;
    avg = (a + b)/2;
    return avg;
}
```

Optional list of parameters

- A comma-separated list of type-name pairs
  - Can't be just types
- Containes formal parameters
- It can be an empty list

Name of the function

- Any valid identifier name

Function body

- Lines of C code evaluating a program logic

Return type of the function

- Any valid data type in C
- `void` in case a function does not return anything explicitly

Another example

```
void print_nums(float a, float b)
{
    prinf("Num1, Num2 are: %d, %d", a, b);
    return;
}
```

Matching Function definition with Function declaration

```
int average(float x,float y);
```

```
int average(float a,float b)
{
    int avg;
    avg = (a + b)/2;
    return avg;
}
```

The following should exactly match between a *Function Declaration* and a *Function Definition*:

- *Name of the Function*
- *Number of Parameters*
- *Type of each Parameter*
  - *Names of the parameters need not match!*
- *The return type*

Function Definitions and Declarations

- Typically, functions are declared first in the program
- Then the functions are defined.
- Although declaring functions is optional, it recommended to declare them before defining them
  - Helps in writing modular programs
  - Typically, functions are declared in ".h" files
    - ".h" files are header files
  - These ".h" files are included in your ".c" files

We will see more about writing modular programs in lab 8

innovate achieve lead

BITS Pilani
Pilani Campus

Function Call

Calling a function

The functions which you have declared and (or) defined can be called from either main() function or any other function.

Example: Function "sum" being called from "main" function

```
/* myProg.c */
#include <stdio.h>
int sum(int a, int b);
int sum(int a, int b)
{
        int total;
        total = a + b;
        return total;
}
int main()
{
        int x,y,z;
        x = 5, y = 4;
        z = sum(x,y);
        printf("Sum is %d:",z};
}
```

Function Declaration

Function Definition

Function Call

Calling a function (another example

```
#include <stdio.h>
int sum(int, int);
int sum_call(int, int);
int sum(int a, int b) {
      int total;
      total = a + b;
      return total;
}
int sum_call(int m, int n)
{
      return sum(m,n);
}
```

```
int main() {
   int x,y,z;
   x = 5, y = 4;
   z = sum_call(x,y);
   printf("Sum is %d:",z};
}
```

- *Function "sum" being called from "sum_call" function*
- *Function "sum_call" being called from "main" function*

Calling a Function (Syntax)

```
function_name (arguments values);
```

Function Name

- This should match with the name used in the function definition and declaration

Argument values

- comma-separated list of expressions
- contains actual parameters

Examples:

```
sum(a, b);
sum(a+b, c+d);
```

Matching Function Call with Function Definition

```
int sum(int a, int b);
int sum(int a, int b){
  int total;
  total = a + b;
  return total;
}
```

```
int main(){
  int x,y,z;
  x = 5, y = 4;
  z = sum(x,y);
  printf("Sum is %d:",z};
}
```

The following should exactly match between a *Function Definition* and a *Function Call*:

- The **Name of the function**
- The **number of arguments** in the function call must match the **number of parameters** in the function definition
- **Type of each argument** in the function call must match with the **type of the corresponding parameter** in the function definition
  - *Names of parameters need not match!*
- The **return type**

Flow of program execution

```
/* myProg.c */
#include <stdio.h>
int sum(int a, int b);
int sum(int a, int b){
     int total;
     total = a + b;
     return total;
}
```

```
int main(){
   int x,y,z;
   x = 5, y = 4;
   z = sum(x,y);
   printf("Sum is %d:",z};
}
```

Pros and Cons of using Functions

Advantages

- **Modularization**
- **Code Reusability**
- **Reduced Coding time**
- **Easier to Debug**
- **Easier to understand**

Disadvantages

- **Reduced execution speed**
  - *Every function call adds an additional overhead to the OS to create space for it in the program memory.*
  - *This slows down the program.*

innovate achieve lead

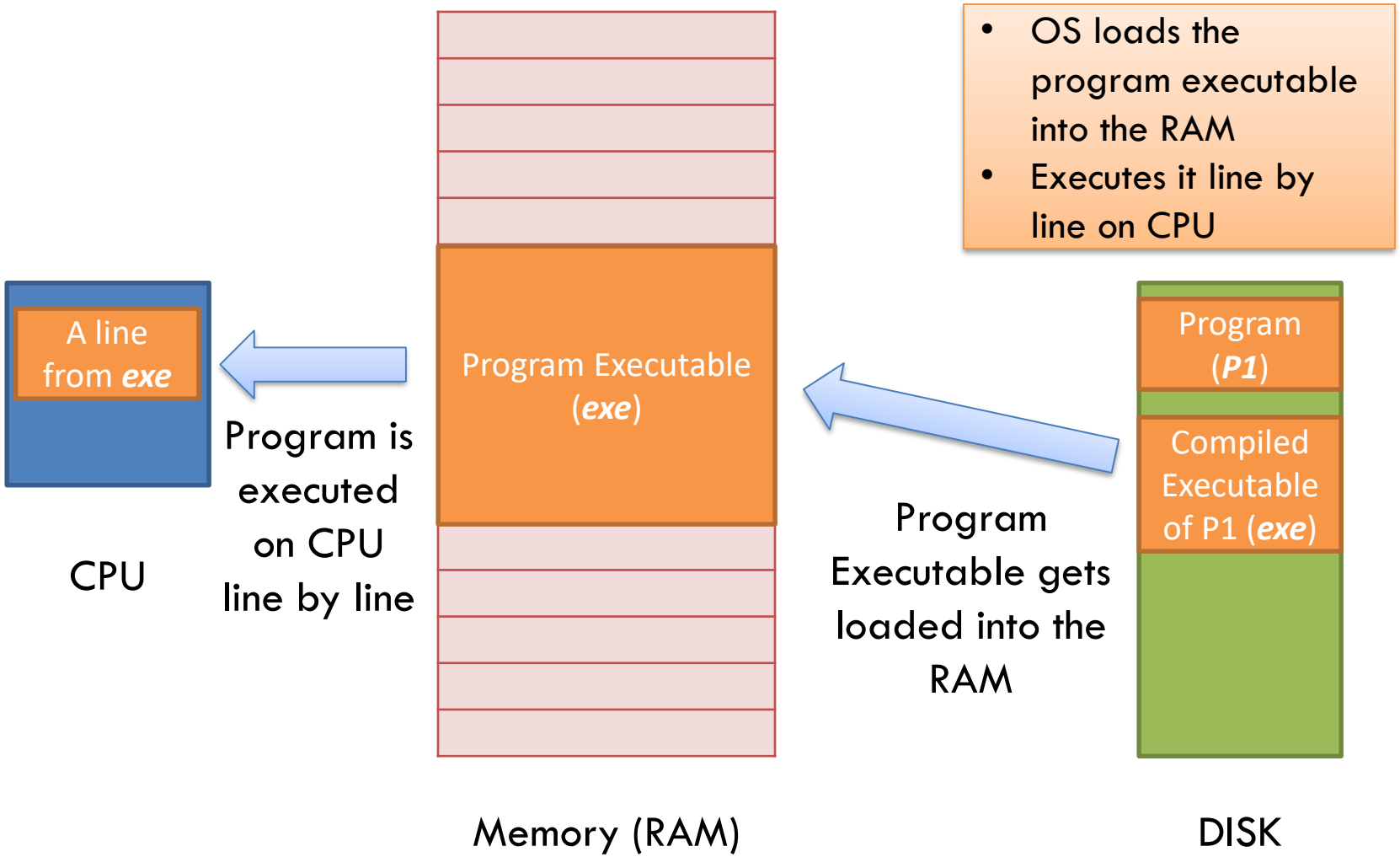
BITS Pilani
Pilani Campus

Functions in Memory

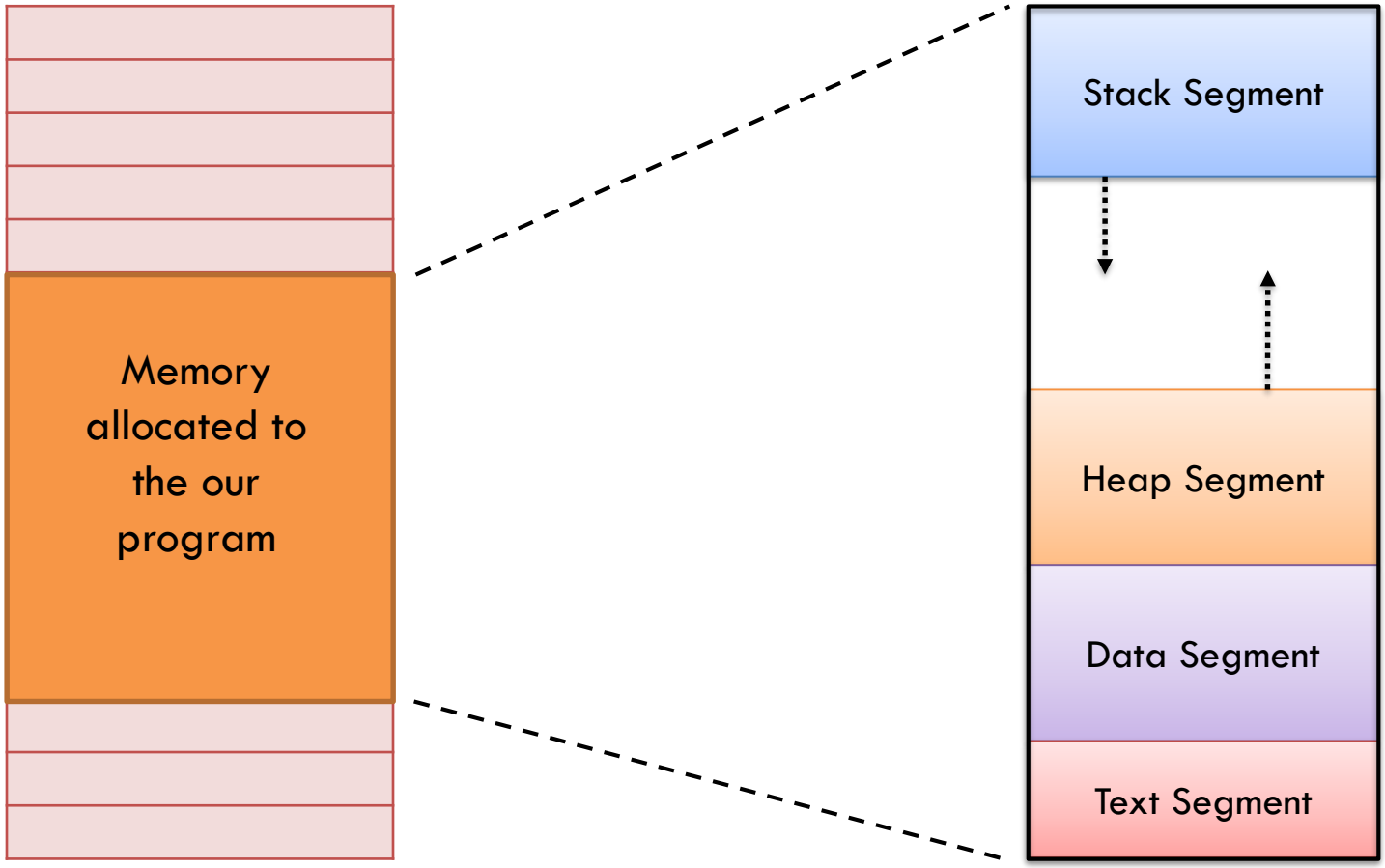
How does function execution look like in memory?

Let us recall our block diagram

Our block diagram is back again!

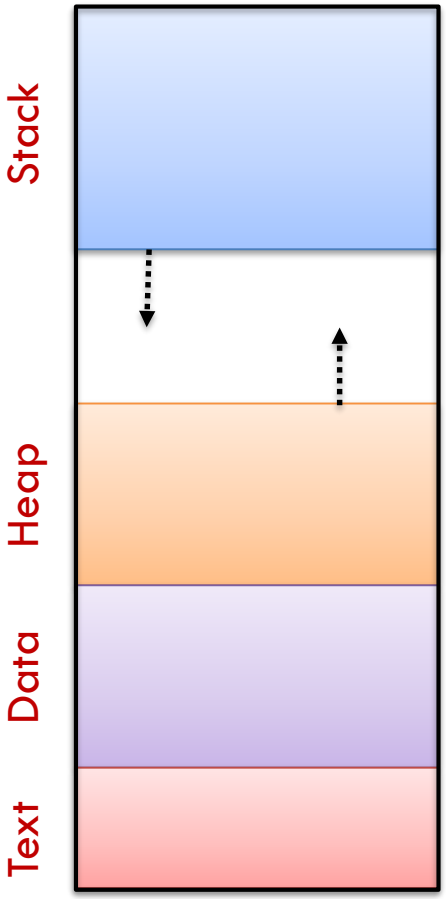


Looking at the main memory only



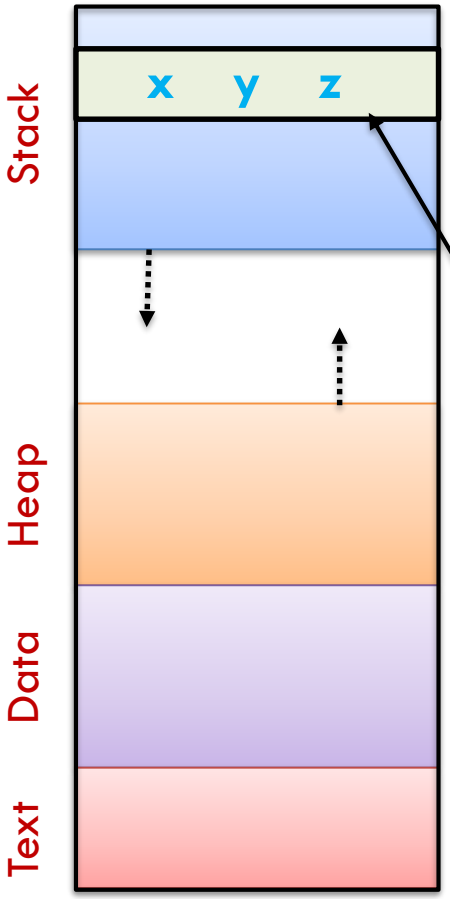
Functions (and variables defined in them) reside in the stack segment of the memory

Functions in Memory




```
/* myProg.c */
#include <stdio.h>
int sum(int a, int b);
int sum(int a, int b)
{
   int total;
   total = a + b;
   return total;
}
int main()
{
   int x,y,z;
   x = 5, y = 4;
   z = sum(x,y);
   printf("Sum:%d",z};
}
```

Functions in Memory

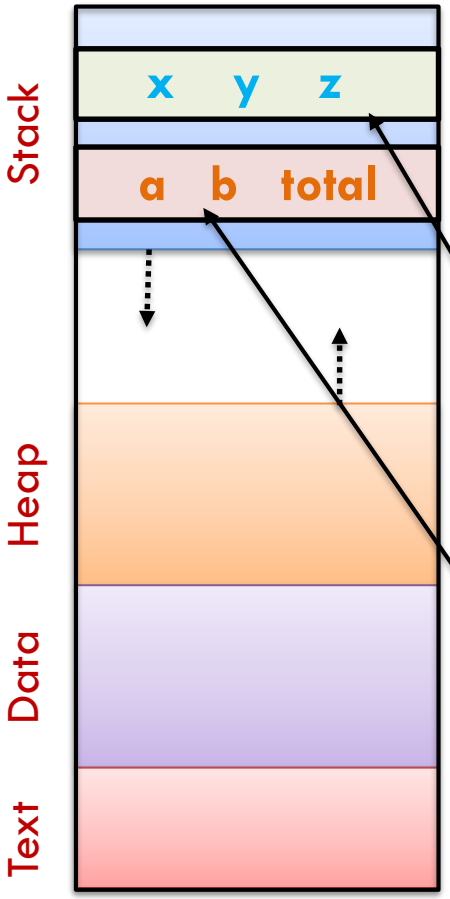


Frame allocated to *main()* in the stack

Variables x, y and z reside inside the stack frame allocated to the function *main()*

```
/* myProg.c */
#include <stdio.h>
int sum(int a, int b);
int sum(int a, int b)
{
   int total;
   total = a + b;
   return total;
}
int main()
{
   int x,y,z;
   x = 5, y = 4;
   z = sum(x,y);
   printf("Sum:%d",z};
}
```

Functions in Memory



Frame allocated to *main()* in the stack

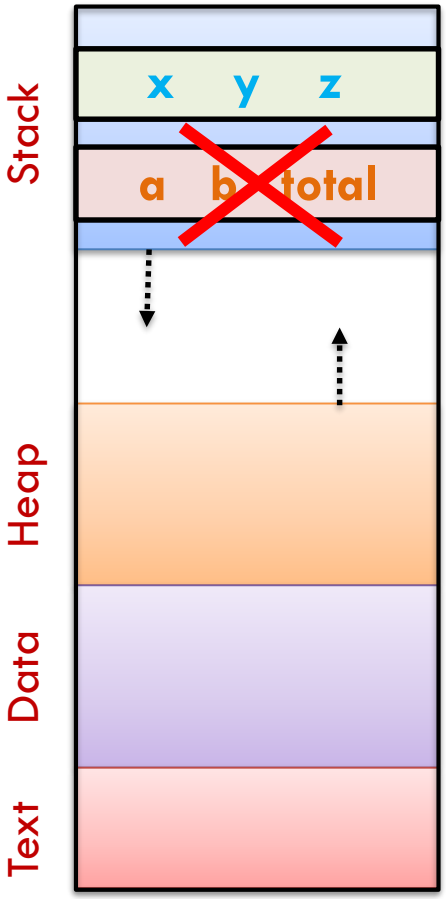
Frame allocated to *sum()* in the stack

Variables x, y and z reside inside the stack frame allocated to the function *main()*

Variables a, b and total reside in the stack frame allocated to the function *sum()*

```
/* myProg.c */
#include <stdio.h>
int sum(int a, int b);
int sum(int a, int b)
{
   int total;
   total = a + b;
   return total;
}
int main()
{
   int x,y,z;
   x = 5, y = 4;
   z = sum(x,y);
   printf("Sum:%d",z};
}
```

Functions in Memory



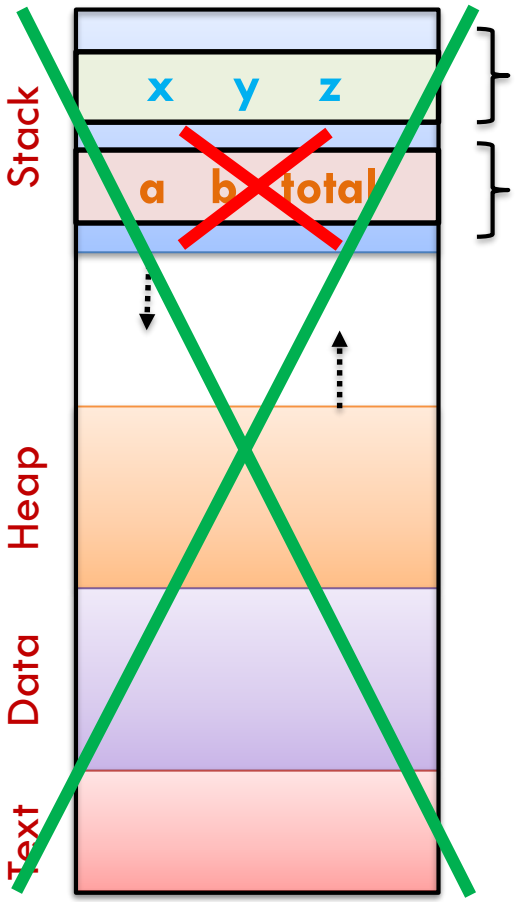
Frame allocated to *main()* in the stack

Frame allocated to *sum()* in the stack

When function *sum()* returns, frame allocated to it is destroyed. So, a, b and total are destroyed

```
/* myProg.c */
#include <stdio.h>
int sum(int a, int b);
int sum(int a, int b)
{
   int total;
   total = a + b;
   return total;
}
int main()
{
   int x,y,z;
   x = 5, y = 4;
   z = sum(x,y);
   printf("Sum:%d",z};
}
```

Functions in Memory



Frame allocated to *main()* in the stack

Frame allocated to *sum()* in the stack

When function *sum()* returns, frame allocated to it is destroyed. So, a, b and total are destroyed.

When function *main()* returns or the program terminates, the entire memory allocated to this program is destroyed.

```
/* myProg.c */
#include <stdio.h>
int sum(int a, int b);
int sum(int a, int b)
{
   int total;
   total = a + b;
   return total;
}
int main()
{
   int x,y,z;
   x = 5, y = 4;
   z = sum(x,y);
   printf("Sum:%d",z};
}
```

innovate achieve lead

BITS Pilani
Pilani Campus

Call by Value

Swap two numbers and "Call by Value"

```
int main()
{
int x, y;
printf("Enter the numbers:\n");
scanf("%d %d", &x, &y);
swap(x, y);
printf("x=%d, y=%d\n",x,y);
return 0;
}
```

```
void swap(int a, int b)
{
int c = a;
a = b;
b =  c;
printf("a=%d, b=%d\n", a,b);
return;
}
```

Call by value:

- When a function is being called, the values of the parameters from caller function are copied into the parameters of the called function.
- Values of x and y are copied into a and b respectively
- Any change to the values of a and b is not reflected into x and y
- a and b are swapped in the swap() function. This swap is not reflected in x and y
- print statement in swap() function prints the swapped values of a & b
- print statement in main() function prints the old values of x and y

Program Execution:

Enter the numbers:

4 5

a=5, b=4

x=4, y=5

innovate achieve lead

BITS Pilani
Pilani Campus

More Examples and Exercises

Examples (Worked out on board)

Example 1:

Write a function `factn()` which accepts input n and computes n!

Write the relevant code in `main()` to call this function.

Example 2:

Write a C program to compute the sum of the series

1 + 2^2 + 3^3 + … n^n (where n is user input).

Use a function long `compute_sum(int)` to compute the sum of the series, and within it, call another function `long power(int)` to compute the term i^i for each i

Exercise: Execute both the programs and observe the output

```
void f1(int);
int f2();
int main() {
   f1(f2());
   return 0;
}
void f1(int a){
   printf("%d",a);
}
int f2(){
   return 5;
}
```

```
int f1(int);
int f2(int);
int main(){
   printf("%d\n",f1(f2(f1(15))));
   return 0;
}
int f1(int a){
   return f2(a/2);
}
int f2(int a){
   return a>5?a:5;
}
```

innovate achieve lead

BITS Pilani
Pilani Campus

Thank you

Q & A