B. Tech. I CSE (Sem-2)
Data Structures
CS102

Memory Management in C and Stack in Function Call

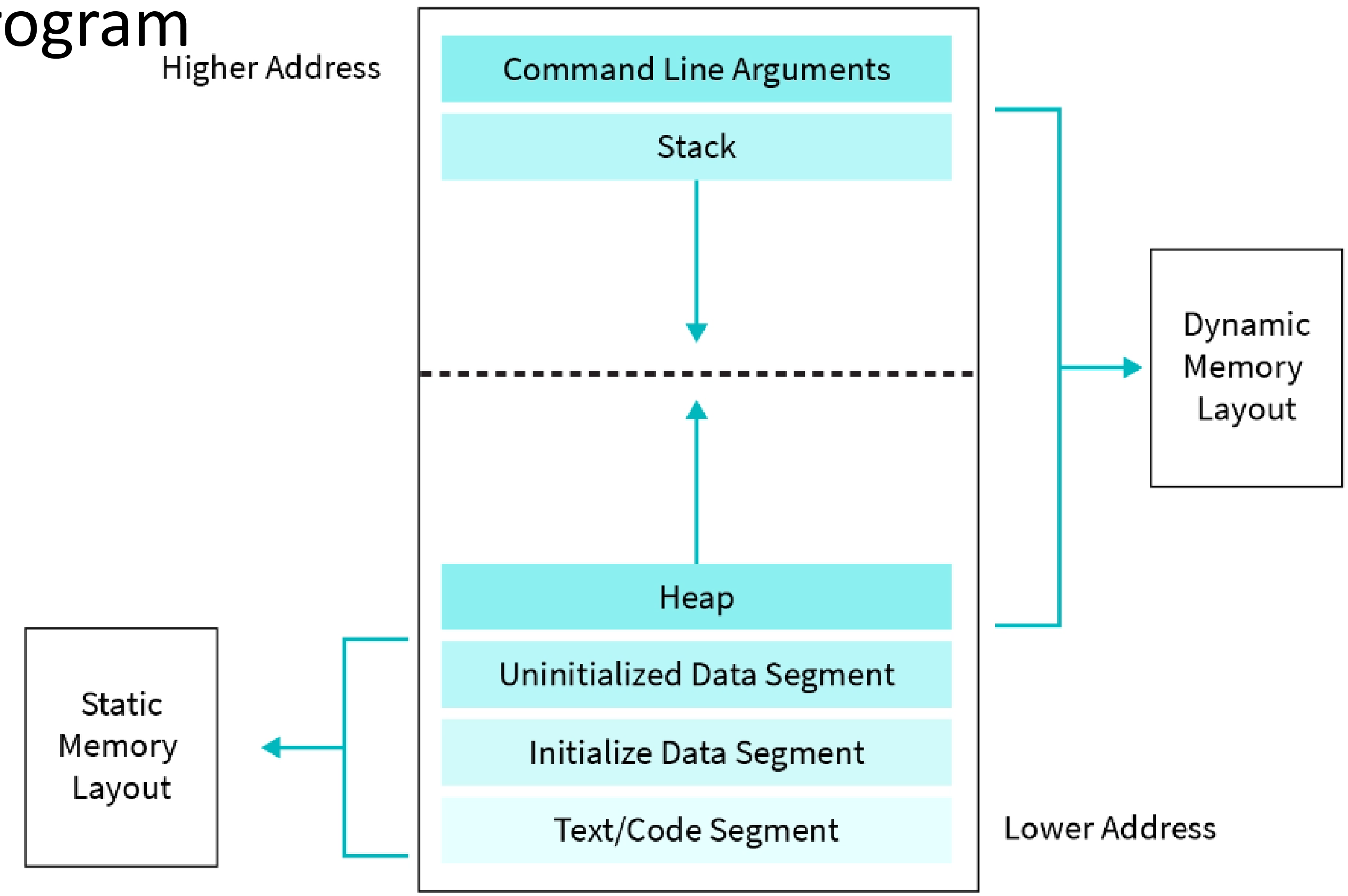
*Some slides are kept with logical or syntax error. So, if you are absent in theory class, please also refer the slides provided at the end of the presentation.

Dr. Rupa G. Mehta
Dr. Dipti P. Rana
Department of Computer Science and Engineering
SVNIT, Surat

C Program Memory Map

Memory representation of C program consists of following sections:

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap



Text Segment

- AKA a code segment or simply as text
- Stores one of the sections of a program in an object file or in memory, which contains executable instructions
- A text segment may be placed below the heap or stack in order
  - To prevent heaps and stack overflows from overwriting it
- Sharable
  - So that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells etc.
- Often read-only
  - To prevent a program from accidentally modifying its instructions

Initialized Data Segment

- Usually called data segment
- A portion of virtual address space of a program
- Contains values of all external, global, static, and constant variables whose values are initialized at the time of variable declaration in the program
  1. initialized read-only area : **const** variable
  2. Initialized read-write area : all other

Example : Initialized Data segment

```
#include <stdio.h>

char c[]="Harry Potter";               /*  global variable stored in Initialized Data
                                        Segment in read-write area*/
const char s[]="HackerEarth";    /* global variable stored in Initialized Data
                                        Segment in read-only area*/

int main()
{
    static int i=11;          /* static variable stored in Initialized Data Segment*/
    return 0;
}
```

Uninitialized Data Segment (bss)

- Uninitialized data segment, often called the “bss” (**Block Started by Symbol**) segment
  - Named after an ancient assembler operator that stood for “block started by symbol.”
- Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing
- Starts at the end of the data segment
- Contains all global variables and static variables
  - That are initialized to zero or
  - Do not have explicit initialization in source code

Example : Uninitialized Data segment (BSS)

```
#include <stdio.h>

char c;              /* Uninitialized variable stored in bss*/

int main()
{
    static int i;     /* Uninitialized static variable stored in bss */
    return 0;
}
```

Heap

- Dynamic memory allocation usually takes place
- Begins at the end of the BSS segment and grows to larger addresses from there
- Managed by malloc, realloc, and free

- Will discuss more with Dynamic Data Structure

Stack

- Called an execution stack or machine stack
- Traditionally adjoined the heap area and grew the opposite direction
- When the stack pointer met the heap pointer, free memory was exhausted
  - With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow opposite directions
- Contains the program stack, a LIFO structure, typically located in the higher parts of memory
  - On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction

Stack

- Automatic variables are stored
- The newly called function then allocates room on the stack for its automatic and temporary variables

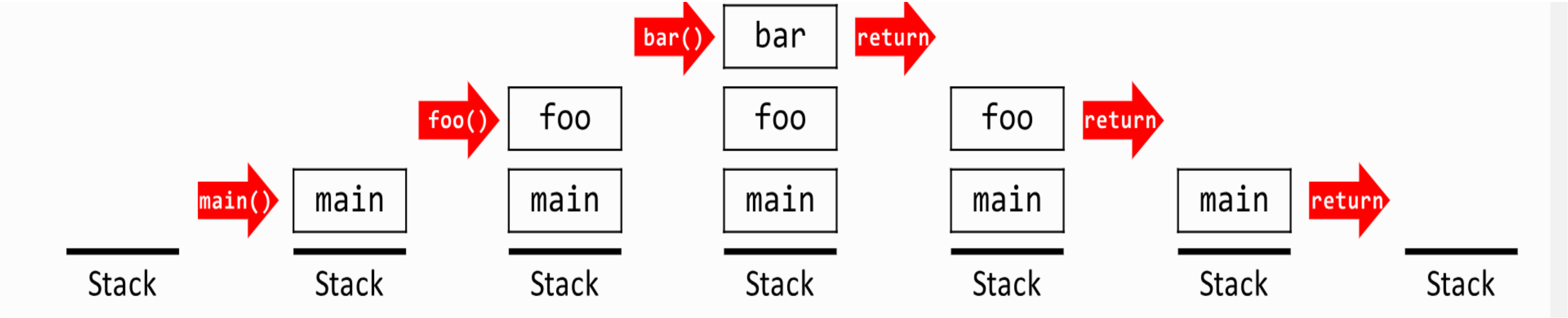
Stack

- A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack.
- The set of values pushed for one function call is termed a “stack frame” or Activation Record
  - A stack frame consists at minimum of a return address
- Activation record contains
  - Local Variables
  - Return address (called function needs to return to the calling function)
  - Certain information about the caller’s environment
    - Such as some of the machine registers

Function call Example

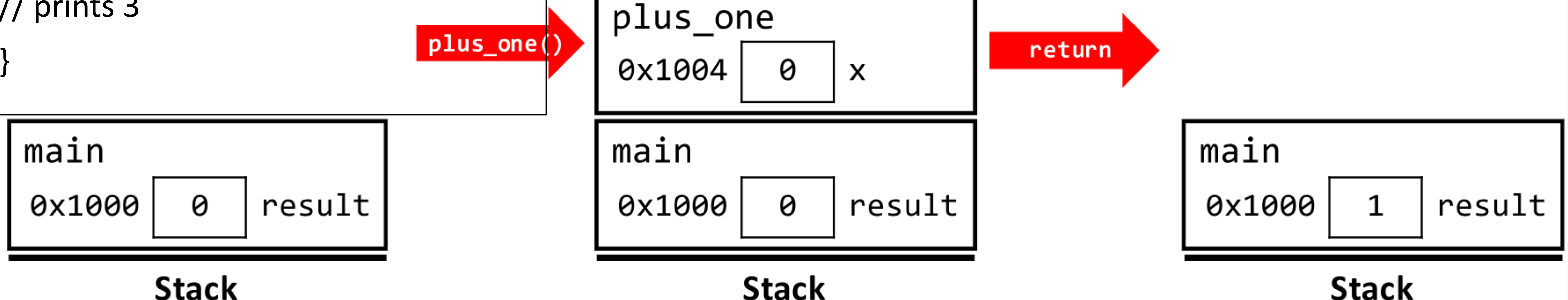
```
void bar() { }

void foo() {  bar();}

int main() {  foo(); }
```

1. When the program is run, the main() function is called
2. Activation record is created and added to the top of the stack
3. main() calls foo() → places an activation record for foo() on the top of the stack
4. bar() is called → so its activation record is put on the stack
5. bar() returns → its activation record is removed from the stack
6. foo() completes → removing its activation record
7. main() returns → the activation record for main() is destroyed



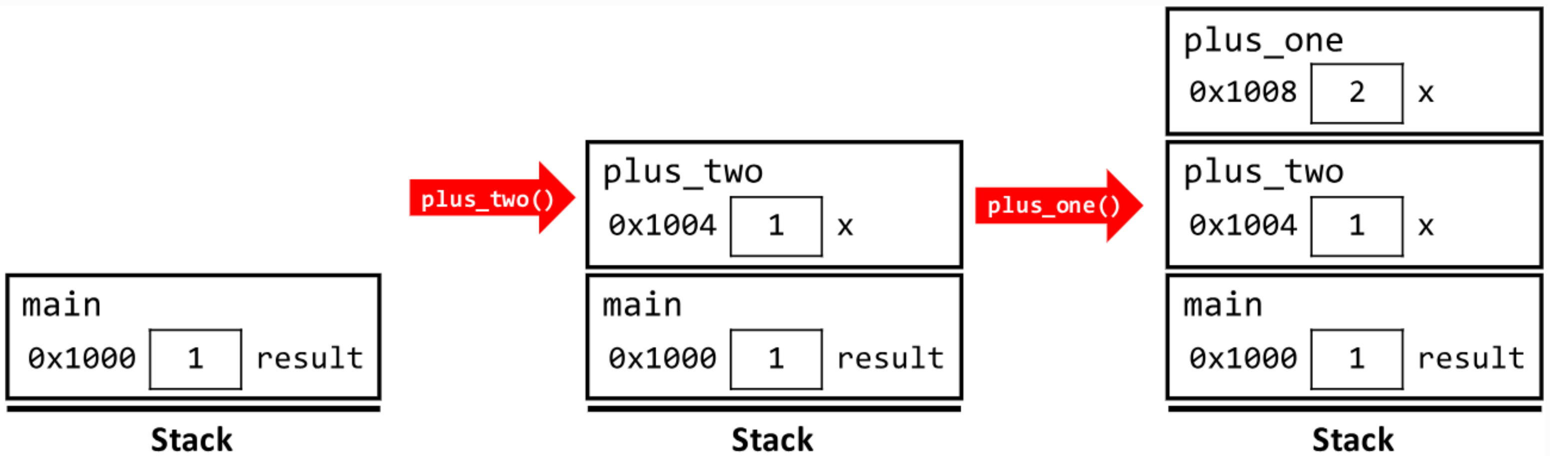
```
6. int plus_one(int x) {
7.  return x + 1;
}
8. int plus_two(int x) {
9.  return plus_one(x + 1);
}
1. int main() {
2.  int result = 0;
3.   result = plus_one(0);
 4.  result = plus_two(result);
 5.  printf("%d"", result);
// prints 3
}
```

1. Line:1 → OS call the main() function
2. Line 2 → Create activation record to holds local variable 'result'
3. Line 2→ The declaration of result initializes its value to 0,
4. Line 3→ program proceeds to call plus_one(0).
    1. creates an activation record for plus_one() that holds the parameter x.
5. Line 6→ The program initializes the value of x to the argument value 0 and runs the body of plus_one().
6. Line 7 → compute x + 1 and return 1
7. The return value replaces the original call to plus_one(0),
8. Line 3 → the activation record for plus_one is discarded before main() proceeds.
9. Line 3 -> assigns the return value of 1 to result.



```
6. int plus_one(int x) {
7.  return x + 1;
}
8. int plus_two(int x) {
9.  return plus_one(x + 1);
}
1. int main() {
2.  int result = 0;
3.   result = plus_one(0);
 4.  result = plus_two(result);
 5.  printf("%d"", result);  // prints 3
}
```

1. Line 4 → call plus_two(result)
2. Creates an activation record for plus_two() that holds the parameter x
3. Line 8 → X initialized to 1
4. Line 9 → body of plus_two() in turn calls plus_one(x + 1).
    1. evaluates x + 1 and x become 2,
5. Lin 6 → creates an activation record for plus_one()
6. initializes the value of x in the new activation record to be 2
7. Line 7 → runs the body of plus_one().

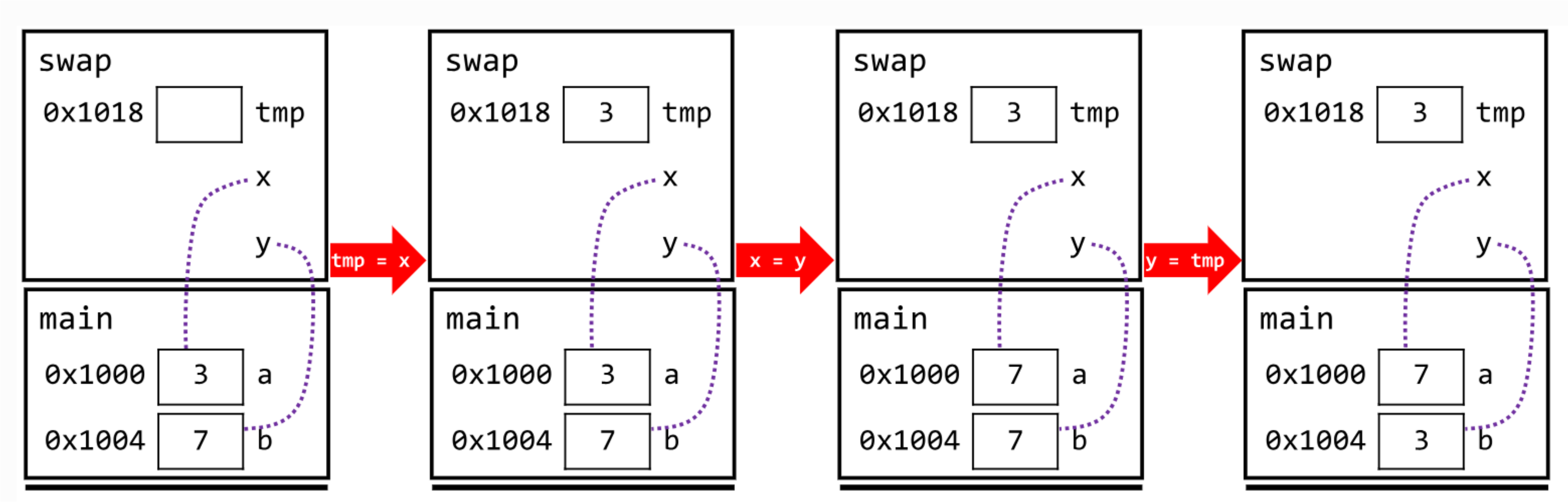


Note:

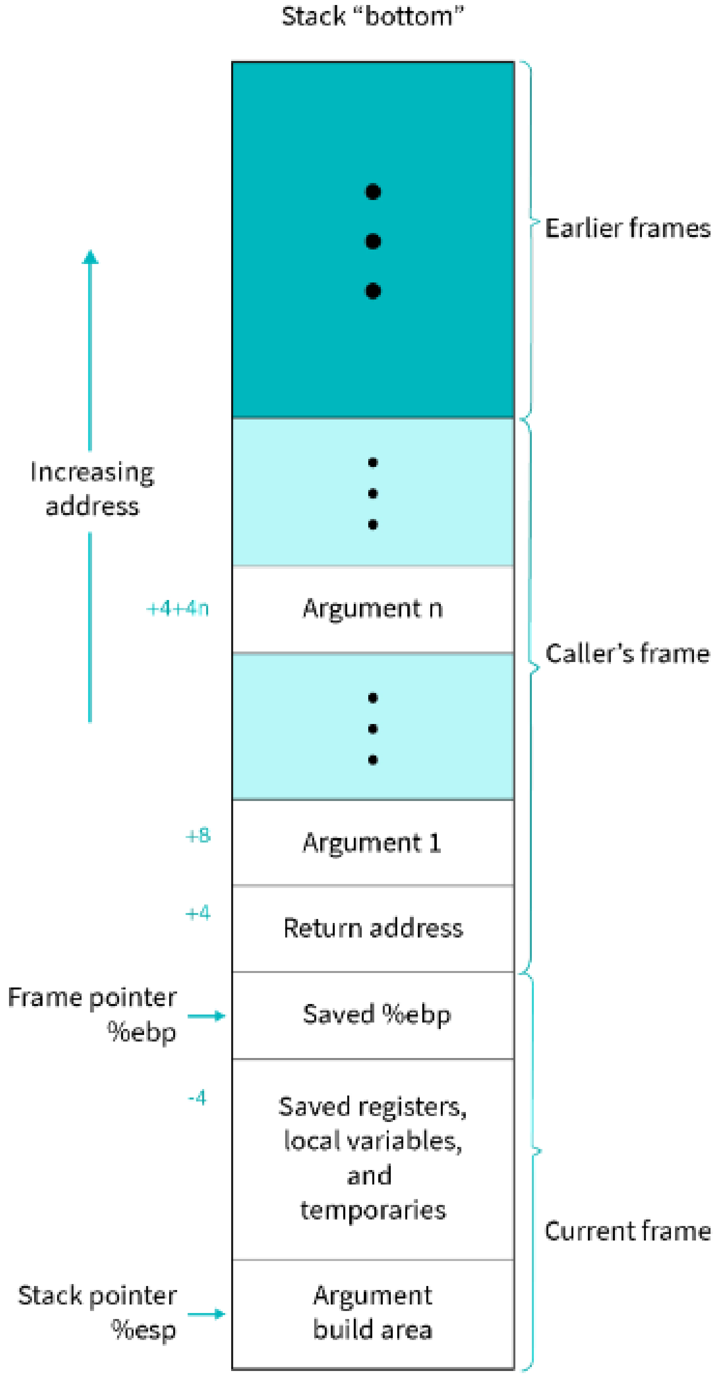
- Main() is also a function
- When main() returns, its stack frame will be popped off and control will be returned to the C runtime system
  - Which is responsible for setting up main() to run in the first place
- The integer return value from main is passed back to whatever entity ran the program in the first place
  - Known as exit status of a program
  - 0 exit status indicates everything went normally for the program
  - Non-zero indicates an error

Call By reference (????)

```
void swap(int &x, int &y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

int main() {
  int a = 3;
  int b = 7;
  printf("%d "d \n",  a, b);
 swap(a, b);
 printf("%d "d \n",  a, b);
}
```



Stack with Multiple Function Calls



Recursion and Stack

- Function call itself to work on a smaller problem
- Each time a recursive function calls itself, a new stack frame is used
- So one set of variables does not interfere with the variables from another instance of the function

Recursion in Stack

- Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively.
- Recursion example
    - Tower of Hanoi, Fibonacci series, Factorial finding, etc.

Example

```
#include <stdio.h>
int fact (int);
int main()
{   int n,f;
    printf("Enter the number whose factorial you want to calculate?");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d",f);
}
int fact(int n)
{   if (n==0)
    {   return 0;   }
    else if ( n == 1)
    {    return 1;  }
    else
    {    return n*fact(n-1);  }
}
```

Example

```c
#include <stdio.h>
int fact (int);
int main()
{   int n,f;
    printf("Enter the number whose factorial you want to calculate?");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d",f);
}
int fact(int n)
{   if (n==0)
    {   return 0;   }
    else if ( n == 1)
    {    return 1;  }
    else
    {    return n*fact(n-1);  }
}
```

Output

Enter the number whose factorial you want to calculate?5
factorial = 120

Example

```c
#include <stdio.h>
int fact (int);
int main()
{   int n,f;
    printf("Enter the number whose factorial you want to calculate?");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d",f);
}
int fact(int n)
{   if (n==0)
    {   return 0;   }
    else if ( n == 1)
    {    return 1;  }
    else
    {    return n*fact(n-1);  }
}
```

Output

Enter the number whose factorial you want to calculate?5
factorial = 120

```
return 5 * factorial(4) = 120
    return 4 * factorial(3) = 24
        return 3 * factorial(2) = 6
            return 2 * factorial(1) = 2
                return 1 * factorial(0) = 1

1 * 2 * 3 * 4 * 5 = 120
```

Fig: Recursion

Algorithm for recursive function

```
if (test_for_base)  then
{
    return some_value;
}
else if (test_for_another_base)
{
    return some_another_value;
}
else
{
    // Statements;
    recursive call;
}
```

- A recursive function performs the tasks by dividing it into the subtasks.
- Termination condition defined in the function:
  - satisfied by some specific subtask
  - After this, the recursion stops and the final result is returned from the function
- base case :
  - The case at which the function doesn't recur
- recursive case :
  - The instances where the function keeps calling itself to perform a subtask

How recursion use stack ?

```
void fun1(int n)
{
      if (n>0)
      {
         printf("%d",n);
         fun1(n-1)
      }
}

void main()
{
      int x=3;
      fun1(x);
}
```

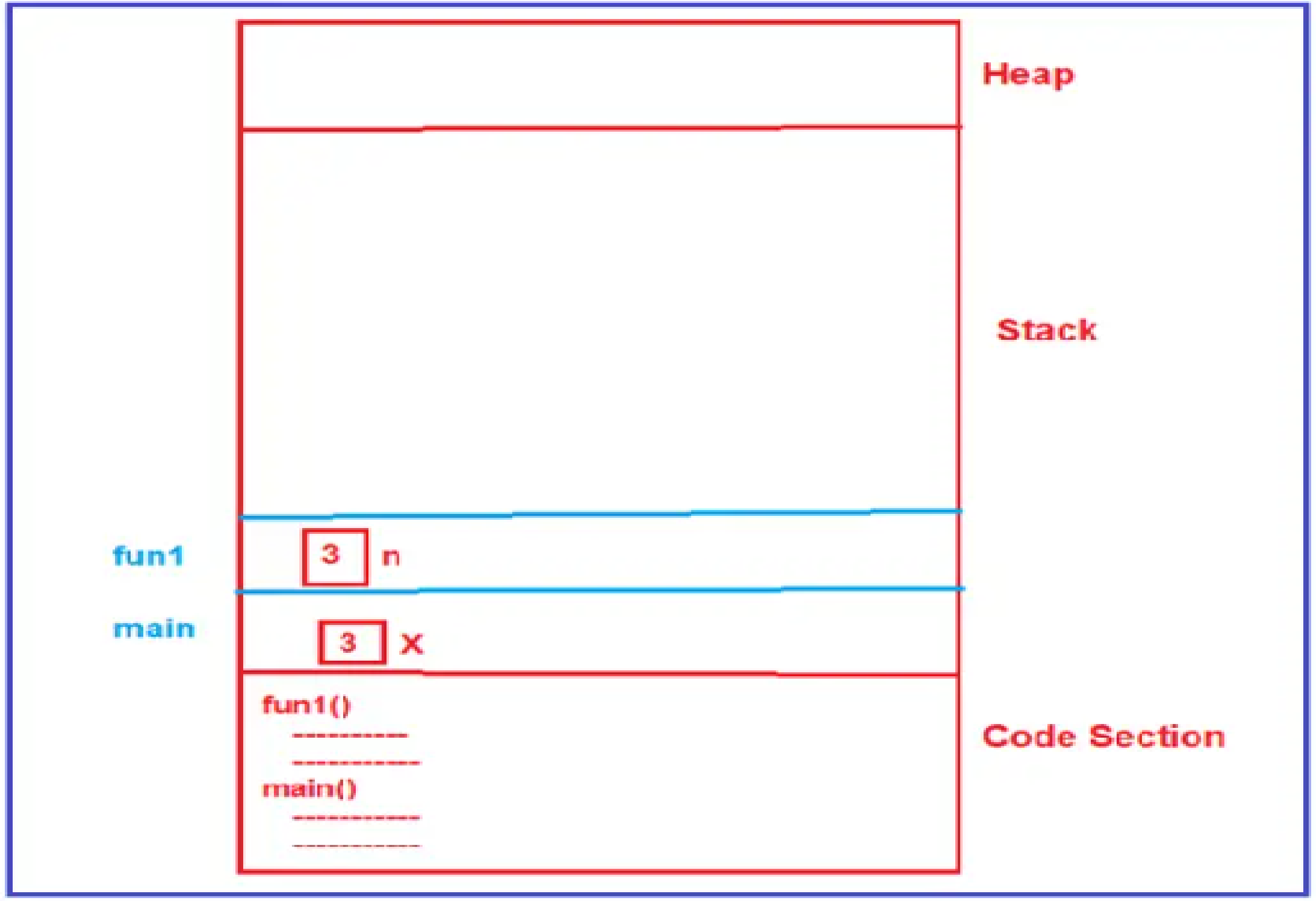
- Each recursive call creates a new copy of that method in the stack
- The machine code of these two functions will be there in the code section of the main memory.
- Now, let us run the program and see how the stack is created.

How recursion use stack ?

```
void fun1(int n)
{
      if (n>0)
      {
          printf("%d",n);
          fun1(n-1)
      }
}

void main()
{
      int x=3;
      fun1(x);
}
```

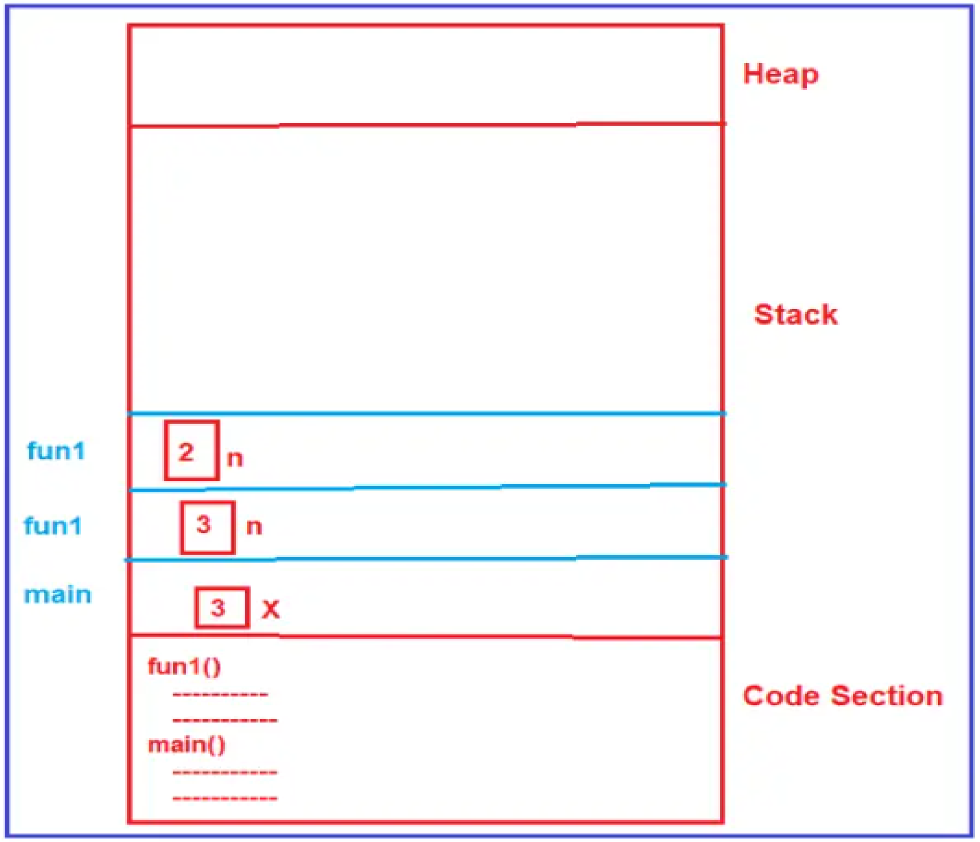
- The program execution starts from the main functions.
- Inside the main function
- int x=3
  - is the first statement that is X is created.
- Inside the stack, the activation record for the main function is created
  - With Local variable X = 3

```
void fun1(int n)
{
    if (n>0)
    {
        printf("%d",n);
        fun1(n-1)
    }
}

void main()
{
    int x=3;
    fun1(x);
}
```



- Next statement is fun1()
  - Call to the fun1 function
  - one variable: n
- The activation record for that fun1() function is created
- n = 3

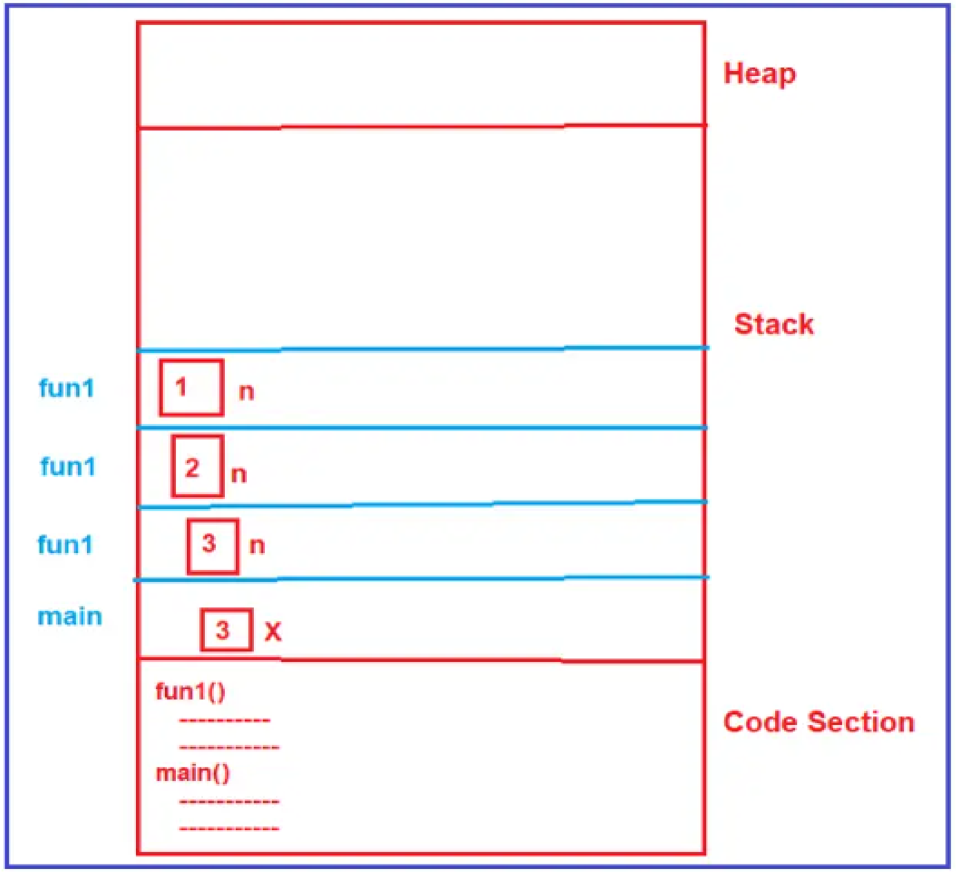
```
void fun1(int n)
{
    if (n>0)
    {
        printf("%d",n);
        fun1(n-1)
    }
}

void main()
{
    int x=3;
    fun1(x);
}
```



Inside the fun1 function

- It will check whether n is greater than 0. Yes, n (3) is greater than 0 and the condition satisfies.
    - So, it will print the value 3
- And will call the fun1() function with the reduced value of n i.e. n-1 i.e. 2.
- Once the fun1 function is called
    - again another activation record for that function is created inside the stack.
- Within this activation record
    - again the variable n is created with the value 2

```
void fun1(int n)
{
    if (n>0)
    {
        printf("%d",n);
        fun1(n-1)
    }
}

void main()
{
    int x=3;
    fun1(x);
}
```

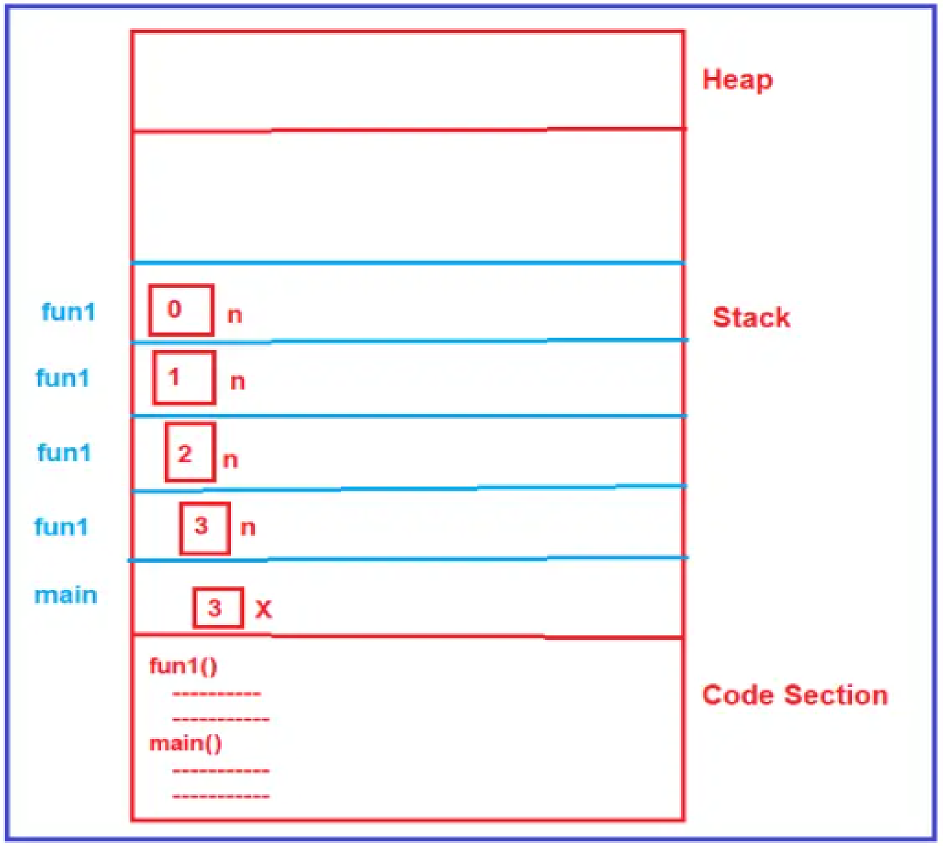


In the second call

- First, it will check whether n is greater than 0. Yes, n (i.e. 2) is greater than 0 and the condition satisfies.
- So, it will print the value 2 and will call the fun1() function with the reduced value of n i.e. 2-1 i.e. 1.
- Once the fun1 function is called
  - another activation record for that function is created (n=1)

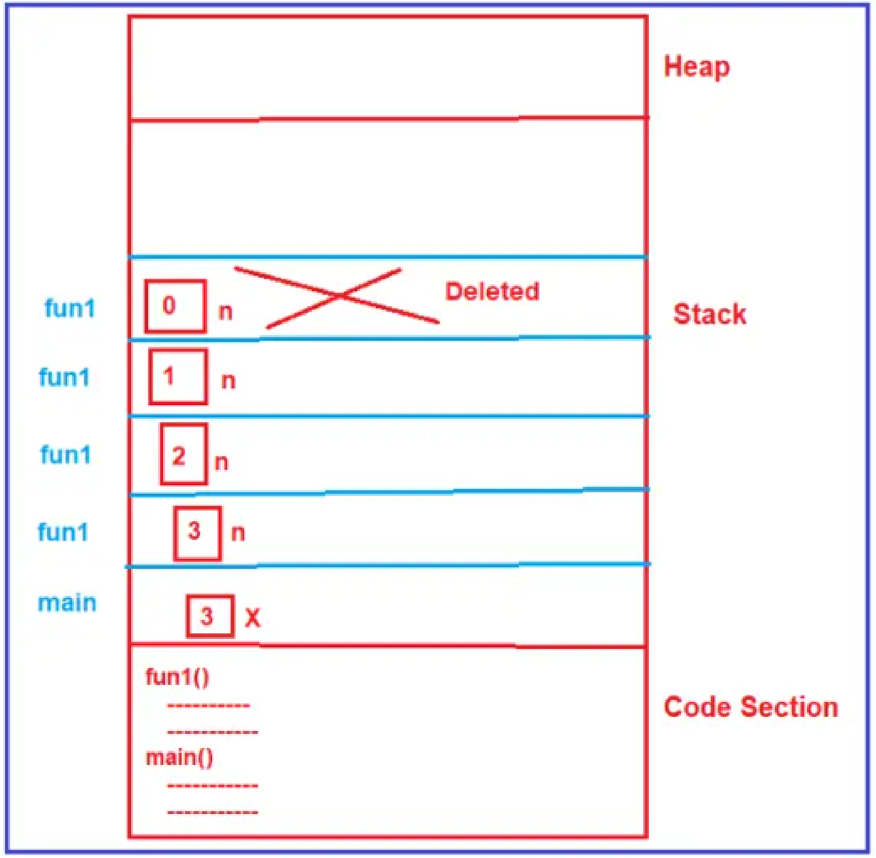
This is the third call of the fun1 function.

```
void fun1(int n)
{
    if (n>0)
    {
        printf("%d",n);
        fun1(n-1)
    }
}

void main()
{
    int x=3;
    fun1(x);
}
```



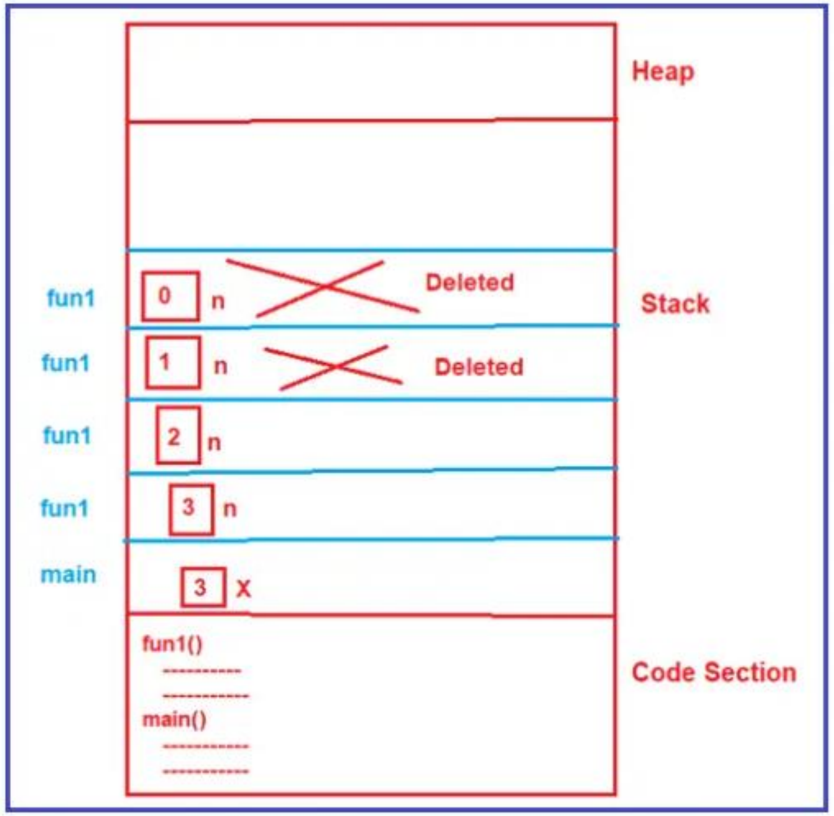
- In the third fun1 function call,
  - It will check whether n is greater than 0. Yes, n (i.e. 1) is greater than 0.
  - So, it will print the value 1 and again it will call the fun1() function with the reduced value of n i.e. 1-1 i.e. 0.
- Once the fun1 function is called,
  - Again another activation record for the fun1 function is created and the variable n is created with the value

```
void fun1(int n)
{
    if (n>0)
    {
        printf("%d",n);
        fun1(n-1)
    }
}

void main()
{
    int x=3;
    fun1(x);
}
```



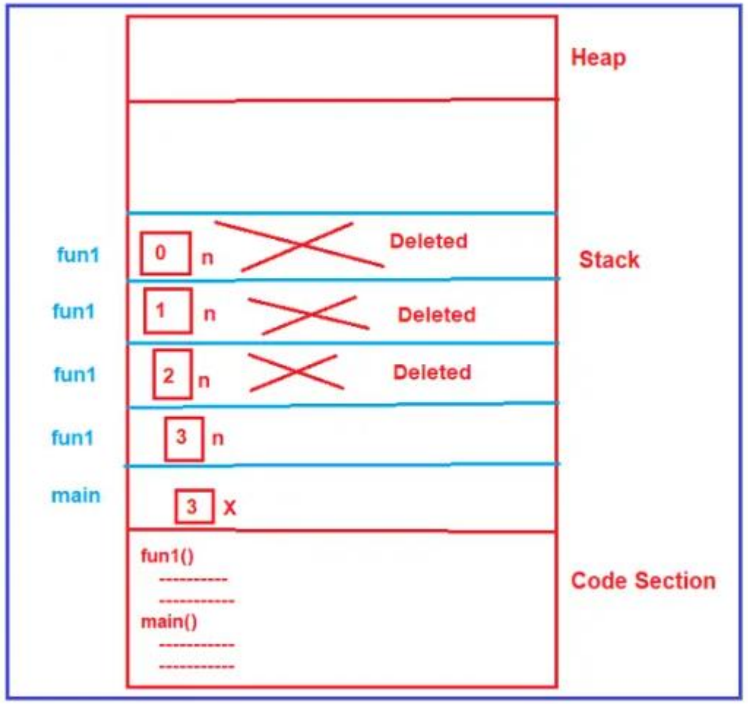
- Now, in the fourth fun1 function call
  - It will check whether n is greater than 0. No, n (i.E. 0) is not greater than 0.
  - So, it will not come inside the condition and comes out of the function.
- Once the fourth fun1 function call completed
  - It will delete that fourth fun1 activation area from the stack

```
void fun1(int n)
{
    if (n>0)
    {
        printf("%d",n);
        fun1(n-1)
    }
}

void main()
{
    int x=3;
    fun1(x);
}
```



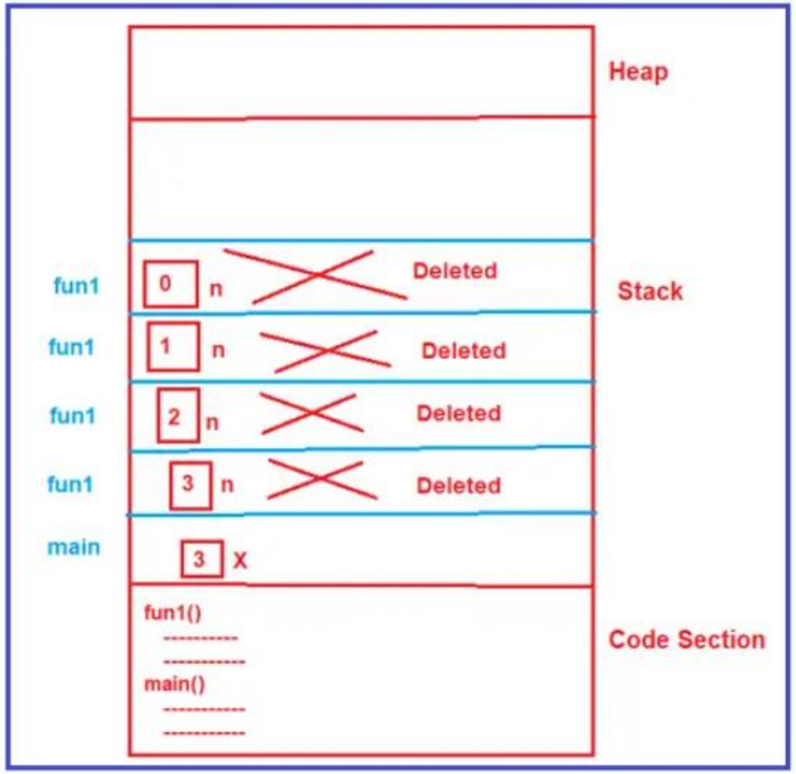
- Once that function call completed
  - And so once that activation record deleted from the stack
  - The control goes back to the previous function call i.E. Fun1(1) i.E. The third function call.
- In the third fun1 function call
  - There are no more operations to perform
- So it simply comes out from that function to the previous function call
  - And also deletes the activation record from the stack

```
void fun1(int n)
{
    if (n>0)
    {
        printf("%d",n);
        fun1(n-1)
    }
}

void main()
{
    int x=3;
    fun1(x);
}
```



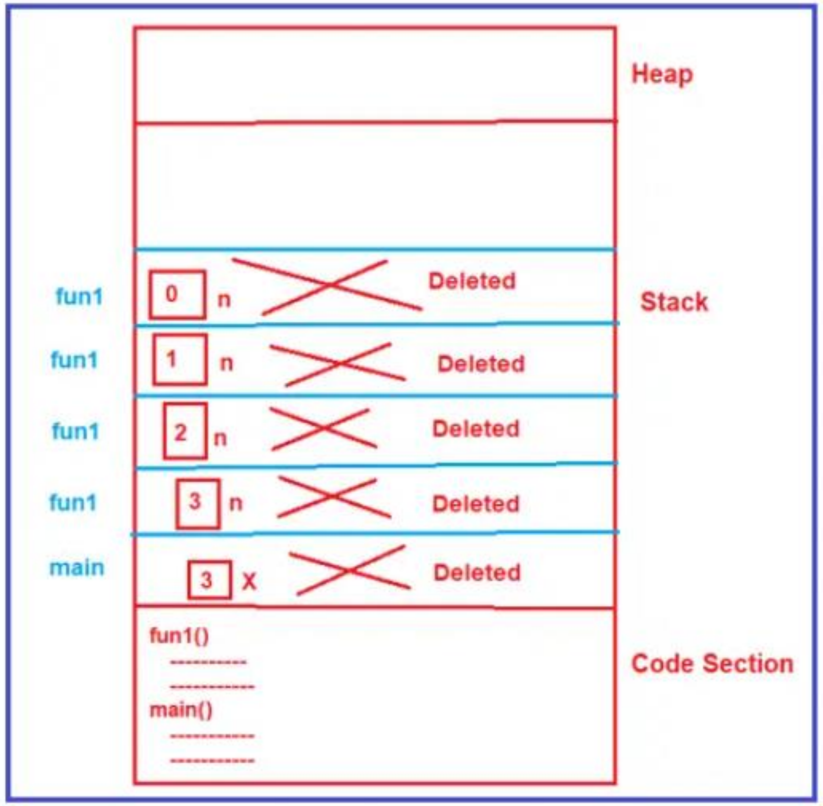
- Once That activation record deleted from the stack,
  - The control goes back to the previous function call i.E. Fun1(2) i.E. The second function call
- In the second fun1 function call, there are no more operations to perform
  - So it simply comes out from that function to the previous function call
  - And also deletes the activation record from the stack which is created for the second function call

```
void fun1(int n)
{
    if (n>0)
    {
        printf("%d",n);
        fun1(n-1)
    }
}

void main()
{
    int x=3;
    fun1(x);
}
```



- Once That activation record for the second function call is deleted from the stack
  - The control goes back to the previous function call i.E. Fun1(3) i.E. The first function call.
- In the first fun1 function call, there are no more operations to perform
  - So it simply comes out from that function to the main function
  - And also deletes the activation record from the stack which is created for the first function call

```c
void fun1(int n)
{
    if (n>0)
    {
        printf("%d",n);
        fun1(n-1)
    }
}

void main()
{
    int x=3;
    fun1(x);
}
```



- Inside the main function after the fun1 function call,
- There is nothing, so it will also delete the activation record which is created for the main function

What is the size of the stack?

- Leaving the main function activation record
  - In our example Total number of Activation function = 4
- So, the size of the stack is 4
  - total size is 4 * size of the variable n
  - In our example Each activation record has single variable n records
- **Note:**
- For x = 3, we have four calls
- If x = 4, then we have 5 calls
- So, for n there will be n+1 calls and so an n+1 activation record

- Recursion uses extra stack memory makes it is memory-consuming functions

Fibonacci Series

int fibonacci(**int**);
main ()
1. display “Enter n“’
2. iInput n
3. f = fibonacci(n);
4. display (f);

int fibonacci (n)
1. **if** (n == 0)
2. **return** 0;
3. **else if** (n == 1)
4. **return** 1;
5. **else**
6. **return** fibonacci(n-1)+fibonacci(n-2);

Advantages and Disadvantages of Recursion in C

- Advantages of recursion:
  1. Writing code may be simpler.
  2. To resolve issues like the Hanoi Tower that are inherently recursive.
  3. Exceptionally practical when using the same solution.
  4. Recursion cuts down on code length.
  5. It helps a lot in resolving the data structure issue.
  6. Evaluations of infix, prefix, and postfix stacks, among other things.

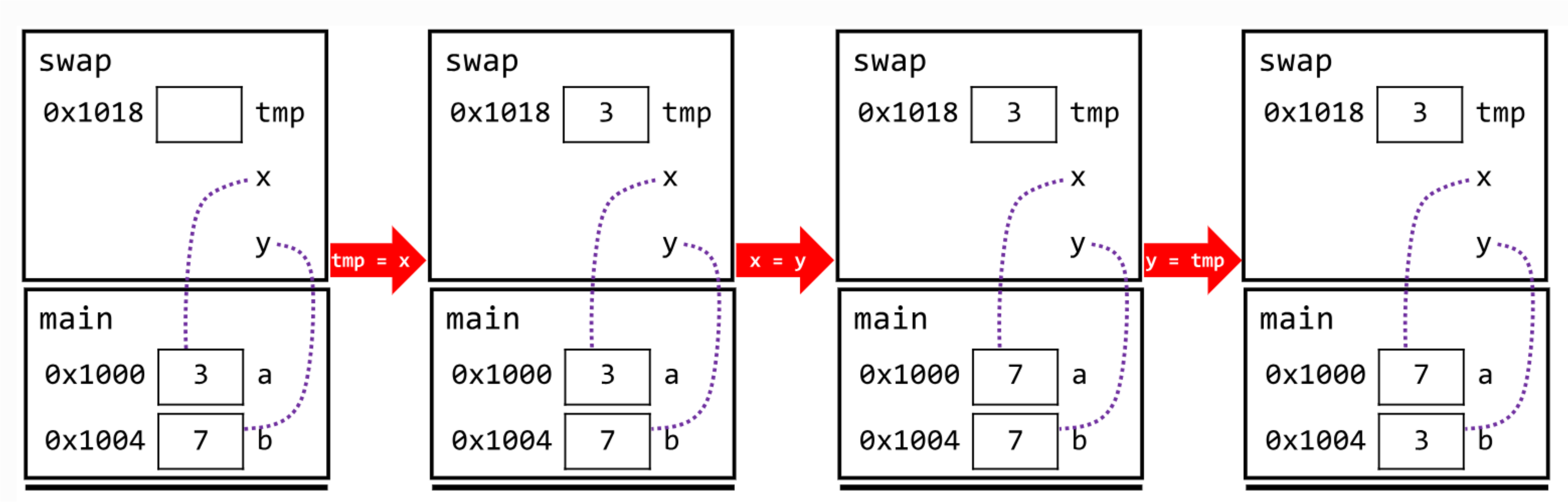
Advantages and Disadvantages of Recursion in C

- Disadvantages of recursion:
    1. In general, recursive functions are slower than non-recursive ones.
    2. To store intermediate results on the system stacks, a significant amount of memory may be needed.
    3. The code is difficult to decipher or comprehend.
    4. It is not more effective in terms of complexity over time and space.
    5. If the recursive calls are not adequately checked, the machine can run out of memory.

Corrected Slides

Call By reference (????): CORRECTED

```
void swap(int *x, int *y)
{
 int tmp = *x;
  *x = *y;
  *y = tmp;
}

int main() {
  int a = 3;
  int b = 7;
  printf("%d %d \n",  a, b);
 swap(&a, &b);
 printf("%d %d \n",  a, b);
}
```



Example: Corrected

```
#include <stdio.h>
int fact (int);
int main()
{   int n,f;
    printf("Enter the number whose factorial you want to calculate?");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d",f);
}
int fact(int n)
{   if (n==0)
    {   return 1;   }
    else if ( n == 1)
    {    return 1;  }
    else
    {    return n*fact(n-1);  }
}
```