Week 8

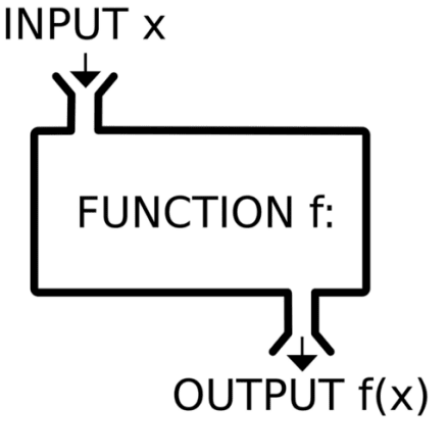
Hari Sethuraman

By the end of today

- You should have a solid understanding of
    - Recursion

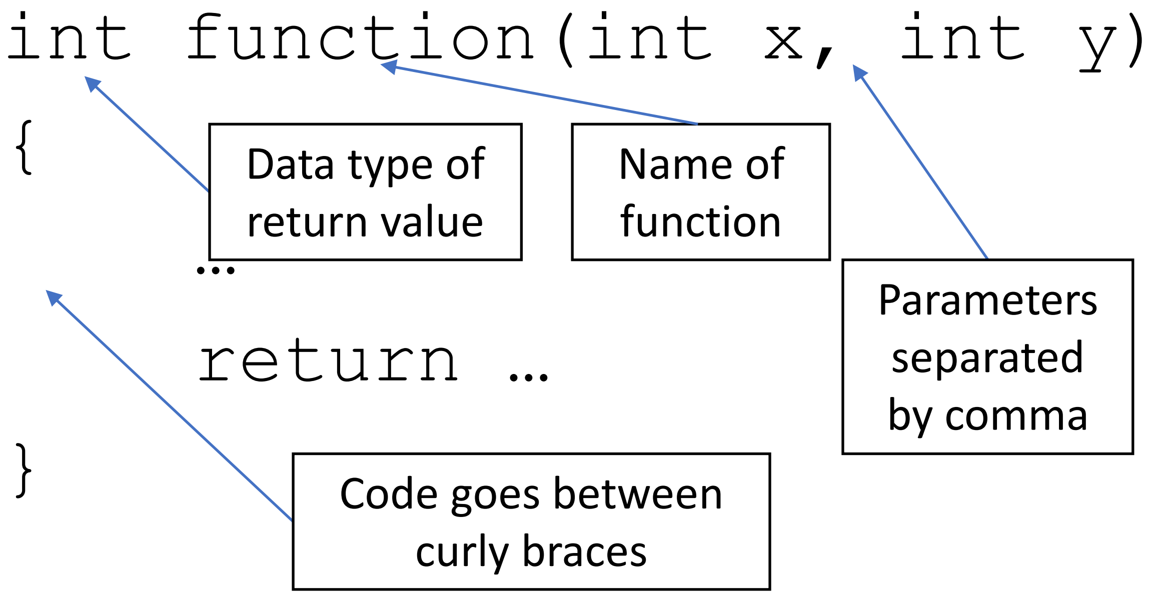
(Recap) What are functions

- Functions are a unit of a program or a building block that take in an input and return a processed value
- Why use functions?
  - Functions allow code to be more reusable and more organized
    - Makes debugging much easier



(Recap) How to declare functions in C

- The input to a function is also called a ‘parameter’
- A function follows certain predefined operations on the parameter
  - You pass in an input value and the function substitutes the parameter with the input value



```
int function(int x, int y)
{
     …
     return …
}
```

Data type of return value

Name of function

Parameters separated by comma

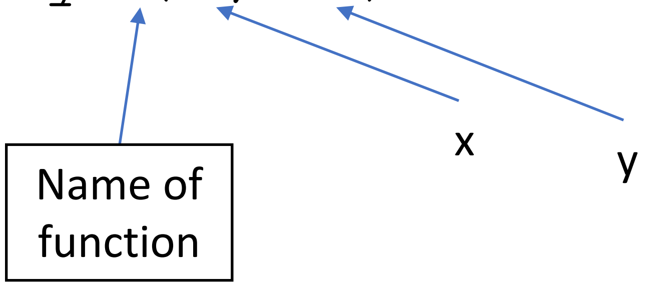
Code goes between curly braces

Always declare functions before you call them in the code!

(Recap) How to call a function

- `Name (parameter1, parameter 2, …);` // parameters in order of declaration!

```
int z = multiply (5, 3);
```



Name of function

x

y

Void functions

- Not all functions need to have a return value
    - Void functions are functions that just do a task without returning a value
- Real-life example of a function that returns a value
    - You give a person a bag of stones (input) and they tell you the number of stones in the bag (output)
- Real-life example of a void function
    - You point at a wall (input) and you ask a person to paint that wall
        - Here the person doesn’t need to return anything, he just needs to paint the wall i.e. do the task.

(Recap) The stack

- The memory used by a program consists of two ‘areas’: the Stack and the Heap
- Is where your computer stores the functions and variables that are called or defined when your program is running
- Works like a stack of trays in a cafeteria. Follows LIFO (Last in first out)
- Every time a new function is called, a new layer is added on the stack which stores the information of the function.
  - All functions below an executing function are frozen (have paused executing)
  - When a function finishes executing it is removed from the stack

(Recap) Program for stack

```
int f3(int n)
{
    int ans = n * n;
} return ans;

int f2(int n)
{
    int ans = f3(n) * 2;
    return ans;
}

int f1(int n)
{
    int ans = f2(n) + 7;
    return ans;
}
int main()
{
   int n = 0;
   scanf(“%d”, &n);
   int a  = f1(n)
   printf(“%d\n”, a);
}
```

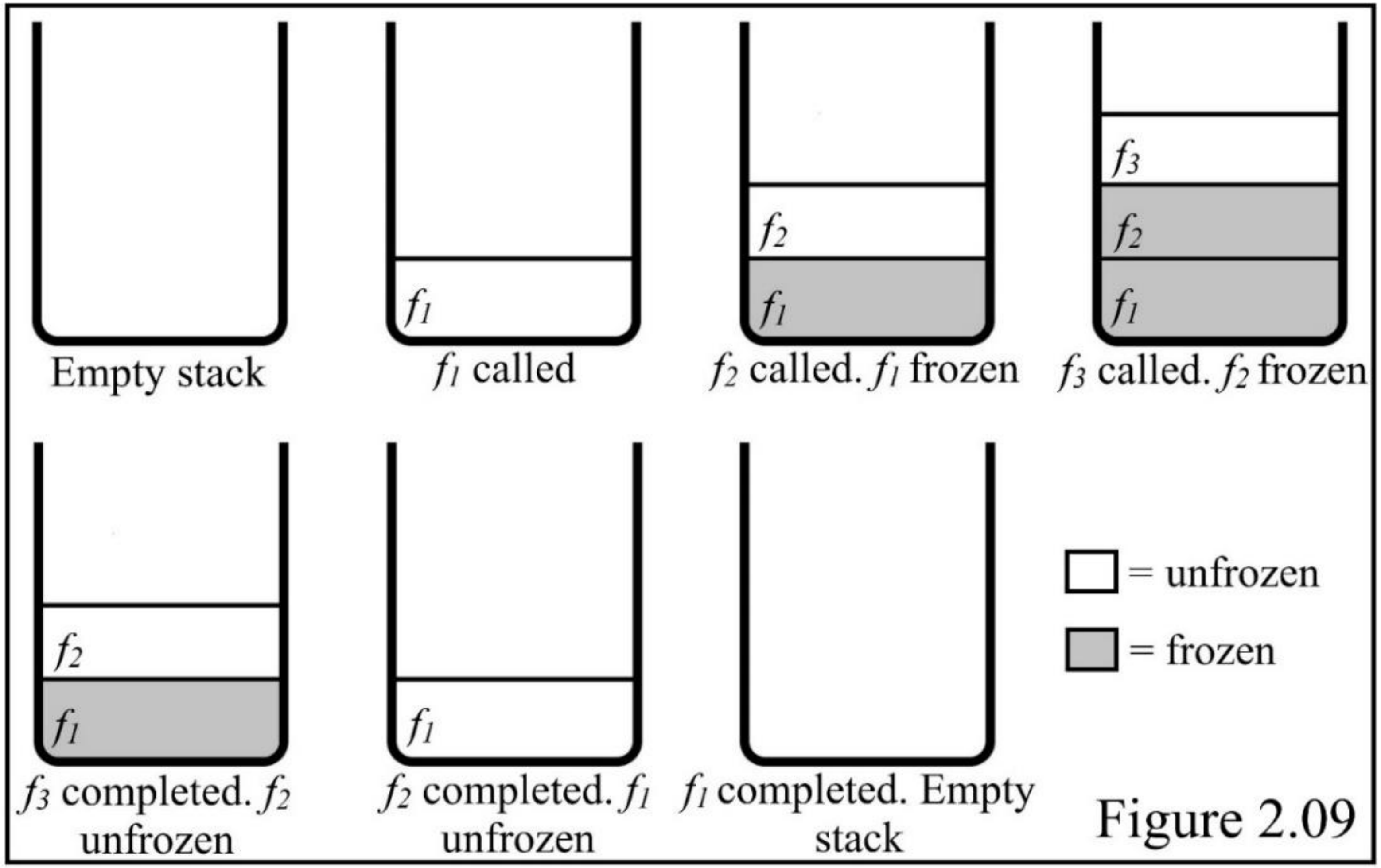

Empty stack
f_1
f_1 called
f_2
f_1
f_2 called. f_1 frozen
f_3
f_2
f_1
f_3 called. f_2 frozen
f_2
f_1
f_3 completed. f_2 unfrozen
f_1
f_2 completed. f_1 unfrozen
f_1 completed. Empty stack
□ = unfrozen
▨ = frozen

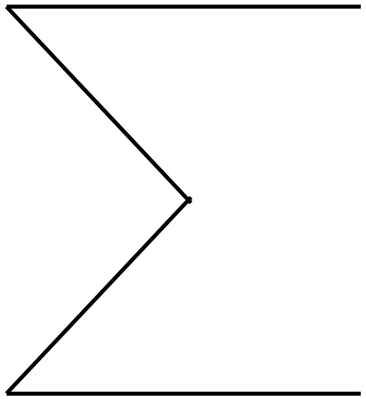
Figure 2.09

Recursion

- Recursive function is a function that calls itself
- Consists of a call and a base case
  - The call is where the function calls itself
  - The base case is a condition that prevents the function from calling itself forever

Sigma

- A function *sigma* takes in a positive integer *n* and returns the sum of all integers up till and including *n*.
- Example: *sigma(5)* = 5
- *Sigma(4) = 10*
- *Sigma(7) = ?*
- *Sigma(9) = ?*



Method 1

- Use a loop that adds all numbers from 1 till *n*

```c
#include <stdio.h>
int main()
{
    int n = 0;
    scanf("%d", &n);
    int sum = 0;
    for (int i = 1; i <= n; i++)
        sum = sum + i;

    printf("%d\n", sum);
}
```

Method 2

- Use recursion
- To solve using recursion, we need to recognize that
  - *sigma(n) = n + sigma(n-1)*
    - Sigma(4) = 4 + sigma(3)
    - Sigma(3) = 3 + sigma(2)
    - Sigma(2) = 2 + sigma(1)
    - Sigma(1) = 1

The call

Base Case

Method 2 In code

```
int sigma(int n)
{
    if (n == 1)
        return 1;
    int sum = n + sigma(n - 1);
    return sum;
}
```

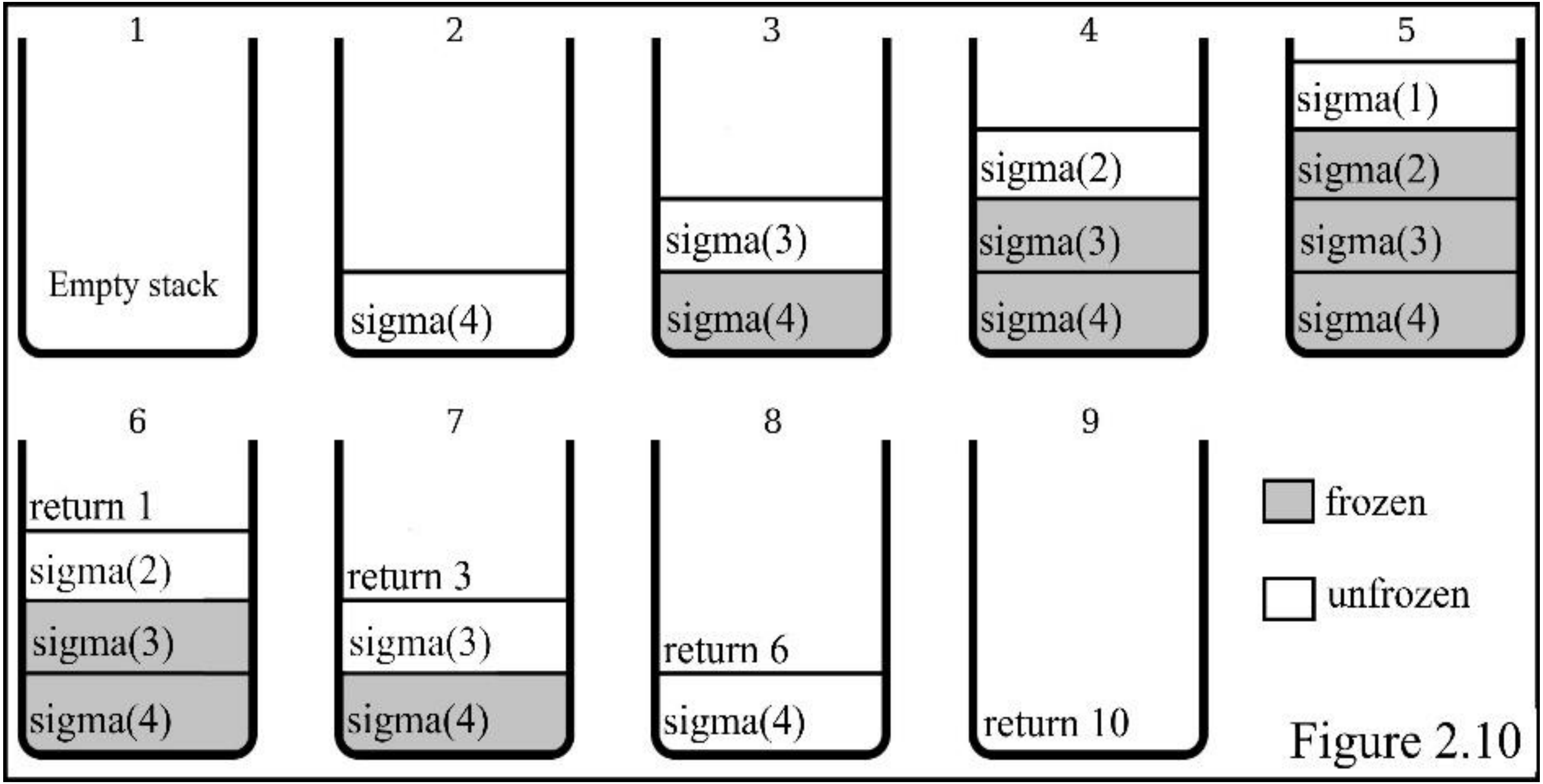

1
Empty stack
2
sigma(4)
3
sigma(3)
sigma(4)
4
sigma(2)
sigma(3)
sigma(4)
5
sigma(1)
sigma(2)
sigma(3)
sigma(4)
6
return 1
sigma(2)
sigma(3)
sigma(4)
7
return 3
sigma(3)
sigma(4)
8
return 6
sigma(4)
9
return 10
▨ frozen
□ unfrozen

Figure 2.10

Method 3

- Sum of positive integers up through *n* can be represented by this formula:
  - Sigma(n) = $\frac{n(n+1)}{2}$
  - Much faster approach as it is just one operation

```
int main()
{
      int n = 0;
      scanf("%d", &n);
      int sum = (n*(n+1))/2;
      printf(“%d\n”, sum);
}
```

The base case

- Always remember to code up the base case
    - Otherwise, the function will call itself for ever, leading to the memory being used up, causing the program to crash.
    - The Base case is almost always the smallest possible value of the input or size of the input
        - If we are dealing with positive integers, the base case would be the smallest value for the number (1)
        - If we are dealing with arrays, the base case would be the smallest length for the array (an array of 1 element)

The call

- The call is a line of code in a recursive function, where the function calls itself.
    - In every consecutive call, the input size or value must approach the base case
        - For arrays, if the base case is triggered when the size of the input array is 1, the length of the array should decrease by a constant factor or number after each consecutive call

Find the largest number in an array

- Given length *n* and an array of integers, find the largest number in the array
- Example
  - n = 6
  - a[] = 1, 6, 23, 9, 43, 4
  - (Max) = 43

Method 1

- Use a loop to iterate over the array

```
printf("Enter num. elements: ");
int length = 0;
scanf("%d", &length);
int nums[length];

for (int i = 0; i < length; i++)
{
    printf("Enter number %d: ", i+1);
    scanf("%d", &nums[i]);
}

int biggest = nums[0];

for (int j = 0; j < length; j++)
{
    if (nums[j] > biggest)
        biggest = nums[j];
}

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

Method 2

- Use recursion to look for the largest element in the array

```
int maxNumInArray(int arr[], int endIndex);

int max(int x, int y);

int main(){

    printf("Hello World\n");

    int n[] = {6, 5, 2, 454, 21, 34};

    int maxNum = maxNumInArray(n, 5);

    printf("%d\n", maxNum);

    return 0;

}

int maxNumInArray(int arr[], int endIndex)

{

    if (endIndex == 0)

        return arr[0];

        return max(arr[endIndex], maxNumInArray(arr, endIndex - 1));

}

int max(int x, int y)

{

    if (x > y)

        return x;

    else

        return y;

}
```