**Session 7**

**Week -3**

## What is Recursion

Recursion is a technique of expressing operations in terms of themselves. In other words, recursion means writing a function that calls itself. Recursion is similar to looping but more powerful. It can make some programs that are nearly impossible to write with loops nearly trivial! Recursion is particularly powerful when applied to data structures such as linked lists and binary trees (which you'll learn about soon). In this chapter and the next, you'll have a chance to understand the basic ideas of recursion as well as see some concrete examples of when you would use it

## Why is Recursion Important?

What are the resource requirements for recursion?
How to think about recursion A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process". This makes it sound very similar to a loop because it repeats the same code, and in some ways it is similar to looping. On the other hand, recursion makes it easier to express ideas in which the result of the recursive call is necessary to complete the task. Of course, it must be possible for the "process" to sometimes be completed without the recursive call.

Example

How would you write a recursive function that prints out the numbers 123456789987654321

| Algorithm: |
| --- |
| |

An Example program with function calls and recursion implementation?

```
using namespace std;
void printNum(int num)
{
	cout << num;
	if (num < 9)
	{
		printNum(num + 1);
	}
	cout << num;
}
int main()
{
	printNum(1);
}
```

Example 2

```
#include<stdio.h>
int findsum(int n);
int main() {
    int num, x;

    printf("\nEnter a number: ");
    scanf("%d", &num);
    x = findsum(num);
    printf("Sum of the digits of %d is: %d", num, x);
    return 0;
}

int r, s;
int findsum(int n) {
    if (n) {
        r = n % 10;
        s = s + r;
        findsum(n / 10);
    }
    else
        return s;
}
```

```
Enter a number: 4
Sum of the digits of 4 is: 4
--------------------------------
Process exited after 3.803 seconds with return value 0
Press any key to continue . . .
```

Example-3

```c
#include<stdio.h>
int main() {
    int pow, num;
    long int res;
    long int power(int, int);
    printf("\nEnter a number: ");
    scanf("%d", &num);
    printf("\nEnter power: ");
    scanf("%d", &pow);
    res = power(num, pow);
    printf("\n%d to the power %d is: %ld", num, pow, res);
    return 0;
}
int i = 1;
long int sum = 1;
long int power(int num, int pow)
{
    if (i <= pow)
    {
        sum = sum * num;
        power(num, pow - 1);
    }
    else
        return sum;
}
```

```
Enter a number: 4

Enter power: 3

4 to the power 3 is: 64
--------------------------------
Process exited after 2.955 seconds with return value 0
Press any key to continue . . .
```

Example 4

```c
#include<stdio.h>
int isPrime(int, int);
int main() {
    int num, prime;
    printf("Enter a positive number: ");
    scanf("%d", &num);
    prime = isPrime(num, num / 2);
    if (prime == 1)
        printf("%d is a prime number", num);
    else
        printf("%d is not a prime number", num);
    return 0;
}
int isPrime(int num, int i) {
    if (i == 1) {
        return 1;
    }
    else {
        if (num % i == 0)
            return 0;
        else
            isPrime(num, i - 1);
    }
}
```

Recursion and data structures

Some data structures lend themselves to recursive algorithms because the composition of the data structure can be described as containing smaller versions of the same data structure. Since recursive algorithms work by making a problem a smaller version of the original, they work well with data structures that are made up of smaller versions of the same data structure—linked lists are one such data structure.

So far we've talked about linked lists as a list onto which you can add more nodes at the front. But another way to think of a linked list is that a linked list is made up of a first node, which then points to another smaller linked list. In other words, a linked list is composed of individual nodes, but each node points to another node that is the start of "the rest of the list".

This matters because it provides a very useful property for us: we can write programs to work with linked lists by writing code that handles either the current node or "the rest of the list". For example, to find a particular node in a list,

Code Snippet (Only)

```
struct node
{
	int value;
	node* next;
};
node* search(node* list, int value_to_find)
{
	if (list == NULL)
	{
		return NULL;
	}
	if (list->value == value_to_find)
	{
		return list;

	}
	else
	{
		return search(list->next, value_to_find);
	}
}
```

The search function solves two possible base cases—we are either at the end of the list or we have found the node we wanted. If neither of the two current cases matches, then we use the search function to solve a smaller version of the same problem. And that's the key—recursion works when you can recursively solve smaller versions of the same problem and use that result to solve the larger problem

**How are recursive functions  different from normal non-recursive functions?**

- A recursive function in general has an extremely high time complexity while a non-recursive one does not.
- A recursive function generally has smaller code size whereas a non-recursive one is larger.
- In some situations, only a recursive function can perform a specific task, but in other situations, both a recursive function and a non-recursive one can do it.

Stack Frame? Activation Records? Dynamic Link?

## How Recursion Works

Two simple and related functions demonstrate how recursion works. The only difference between the two functions is the order of the statements appearing inside the if-statement. This seemingly insignificant difference is sufficient to cause the two functions to print the digits of their parameters in different orders:

- `forward(123)` prints `123`
- `reverse(123)` prints `321`

```
void forward(int number)
{
    if (number != 0)
    {
        forward(number / 10);
        cout << number % 10;
    }
}
```

```
void reverse(int number)
{
    if (number != 0)
    {
        cout << number % 10;
        reverse(number / 10);
    }
}
```

**Figure 1. Simple recursion examples**. Two functions that print the digits of an integer one at a time. Each function satisfies the requirements for recursion:

1. Recursion takes place in the body of the if-statements where the functions call themselves (highlighted).
2. If the test, `number != 0`, is false, recursion does not take place, the function ends and returns.
3. For both functions, the parameter, `number / 10`, changes for each recursive call; eventually the parameter goes to 0, which makes the if-statement false and ends the recursion

One way of understanding recursion is in terms of the stack frames that make automatic variables possible. Each time the program calls a function, it pushes a new stack frame on the runtime stack. Each frame holds the return address and all the local, automatic variables defined in the function (including the function parameters). When the function ends, its stack frame is popped off and discarded, deallocating the memory for the automatic variables. The following picture illustrates the role that stack frames play in recursive function calls.

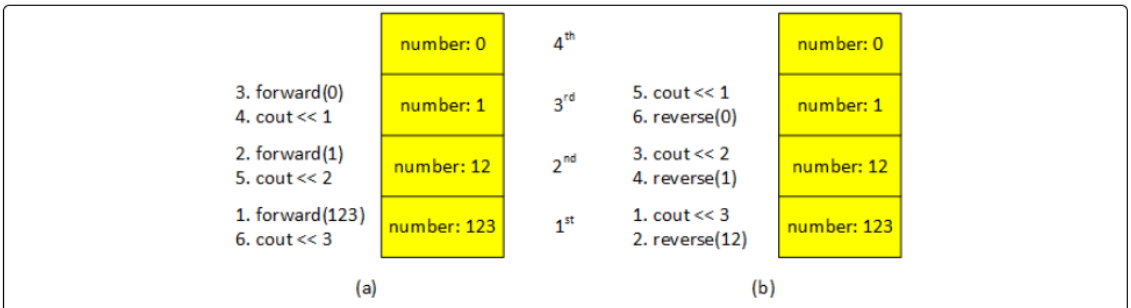



**Figure 2. Recursive function calls and stack frames**. Each yellow box represents a stack frame. The program creates a new stack frame whenever it calls a function; so, when it calls a function recursively, it creates a new stack frame for each call, leaving the previous stack frames intact. In both examples, the program calls the recursive function is called four times. It pushes four stack frames on the stack. Each frame stores one instance of the variable `number`, and each instance holds a different value. The first stack frame is at the bottom, and the last frame is at the top. Numbers on the left side of each stack frame indicate the order in which the statements run.

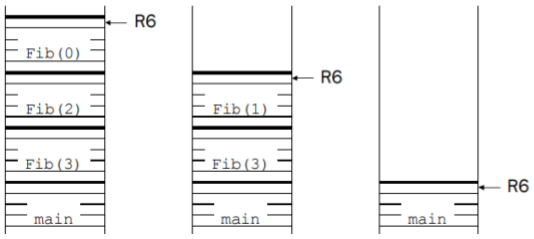
a. The first three statements that run are recursive function calls (1, 2, and 3), which all take place before the output statements. The output statements (4, 5, and 6) are the last to run and they do so after each successive recursive function call returns.

b. The output statements and the recursive function calls are interleaved, but the output statements always run first. As the recursive function call is the last statement inside the if-statement and the if-statement does not have an "else" part, there are no statements to run when a recursive function calls return.

## Activation Record

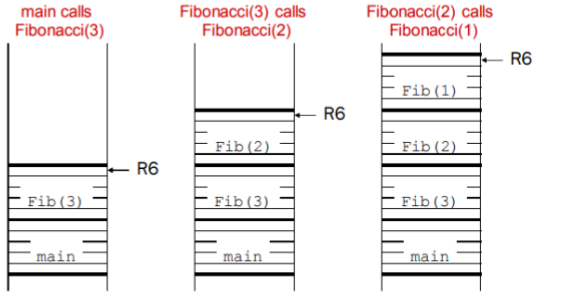
- A recursive function is one that solves its task by calling itself on smaller pieces of data.
  - Similar to recurrence function in mathematics.
  - Like iteration -- can be used interchangeably; sometimes recursion results in a simpler solution.
- Standard example: Fibonacci numbers
  - The n-th Fibonacci number is the sum of the previous two Fibonacci numbers.
  - F(n) = F(n – 1) + F(n – 2) where F(1) = F(0) = 1

```
int Fibonacci(int n){
    if ((n == 0) || (n == 1))
        return 1;
    else
        return Fibonacci(n-1) + Fibonacci(n-2);
}
```




## Activation Records

- Whenever Fibonacci is invoked, a new activation record is pushed onto the stack.




## Anatomy of a Recursive Call,