كلية علوم الحاسوب وتكنولوجيا المعلومات

المرحلة الثانية

مادة الخورازميات

م.م فرح معاذ جاسم

3. Recursion

Recursion is a programming tool that allows the user to write functions that calls themselves.

3.1 Factorial function

Factorial function plays an important role in mathematics and statistics. Given a positive integer, n, factorial is defined as the product of all integers between n and 1. For example 5 factorial equals 5 * 4 * 3 * 2 * 1 = 120 . In mathematics, the exclamation mark (!) is often used to doubt the factorial function. We may write the definition of this function as follows:

$$\left.\begin{array}{l} n! = 1 \\ n! = n * (n-1) * (2-2) * \ldots * 1 \end{array}\right\} \begin{array}{l} \text{if } n == 0 \\ \text{if } n > 0 \end{array}$$

, as an example for this definition:

0! = 1

1! = 1

2! = 2 * 1

3! = 3 * 2 * 1

4! = 4 * 3 * 2 * 1

For any a > 0 we see that n! equals (n-1) !. Multiplying n by the product of all integers from n-1 to / yields the product of all integers from n to 1. We may therefore define:

0! = 1

1! = 1 * 0!

2! = 2 * 1!

3! = 3 * 2 !

4! = 4* 3!

So we an write our mathematical notation as:

n! = 1 if (n = =0)

n! = n (n-1) ! if n > 0

The recursive algorithm to computer (n !) may be directly translated into a C++ function as follows :

Factorial function without recursion

```
int fact(int n)
  {
   int i,prod,fact;
   prod=1;
   for(i=1;i<=n;i++)
   prod*=i;
   fact=prod;
   return fact;
  }
```

Factorial function with recursion

```
  int fact (int n)
  {
  if(n > 1)
  return n * fact(n - 1);
  else
   return 1;
  }
```

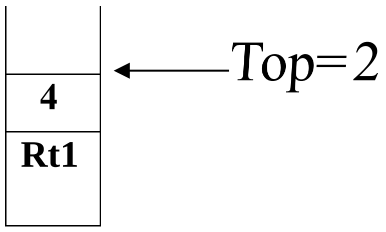
When this function is called in the main program and also with in itself, the compiler uses <u>a stack</u> to keep the successive generations of call variables and parameters. This stack is maintained by the C system and is invisible to the user.

Each time that a recursive function entered, anew allocation of its variables is **pushed on top of the stack**.

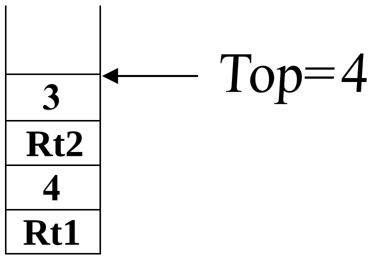
When each call to the subprograms ((recursive procedure) or (recursive function), the push address of the return address (Rt) is stored in the stack, along with copies of the parameter values for that call, and this is repeated at every call to the subprogram until a base case is reached Where the inverse process begins, which is to pop out the contents of the stack sequentially and reach the final result.

We see the state of the stack when we implement it. Expect to calculate the factorial of a number (4), meaning [fact (4)]:

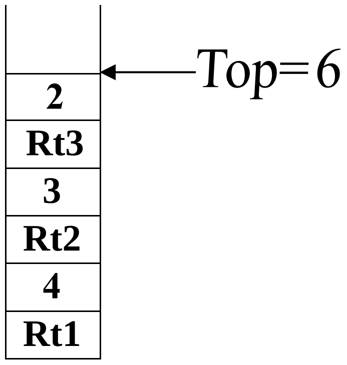
1. At the first call to program (4), the return address (Rt1) and number (4) will be stored in the stack



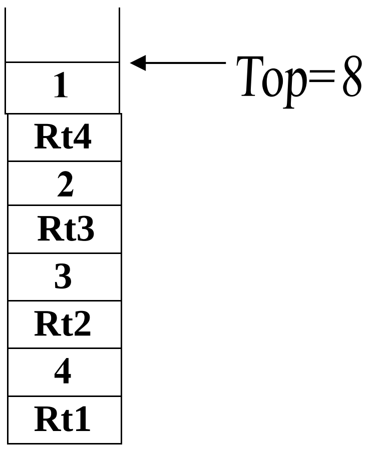
2. In the (else) statement, the function calls the next number (n-1), meaning fact(3), and therefore the reference address for this call (Rt2) and number (3) is stored in the stack.



3. In the subsequent recall of the next number n-1, that is, fact (2) will be stored in the stack, its return address (Rt3) and the number (2).

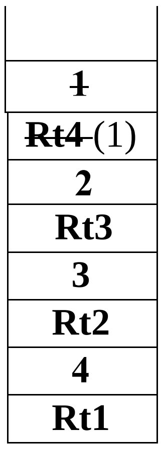


4. The next call is for the next number n-1, which is fact (1), and the return address (Rt4) and number (1) will be stored in the stack.



5. The next call is for the next number n-1, which is fact (1), and the return address (Rt4) and number (1) will be stored in the stack.

6. The (POP) value of the variable (n = 1) is taken out, the return address (Rt4) is output, and the calculation result is entered (1).

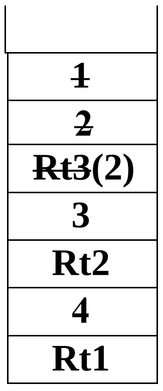


PoP(1)

Rt4=1

Push(1)

7- Output (POP) the last result which is (1) and the variable (n = 2) with the return address (Rt3) to calculate the result of the function which is (2) and then (push) enter this result in the stack.



PoP(1)

PoP(2)

Rt3=2*1

Push(2)

8- Take out (pop) the last result which is (2) as well as the variable (3) with the return address (Rt2) to calculate the result of the function which is 6 = 3 * 2 and then (push) enter this result in the stack.

~~2~~
~~3~~
~~Rt2~~(6)
4
Rt1

PoP(2)

PoP(3)

Rt2=2*3

Push(6)

9- Take out (POP) the last result which is (6), as well as each of the variable (4) with the return address (Rt1) to calculate the function, which is 24 = 4 * 6, then enter (push) this result in the stack.

24
Rt1

PoP(6)

PoP(4)

Rt1=6*4

Push(24)

10 - The only value remaining in the stack is the final result of the factorial of the number, which is 24 because n! = 4! = 4 * 3 * 2 * 1

3.2 Power Function

Power function in C++ without recursion

```
int power(int x,int m)
{
 int p,i,power;
 p=1;
 if(m!=0)
 for(i=1;i<=m;i++)
 p*=x;
 power=p;
 return power;
}
```

With recursion

```
int power1(int x,int m)
{
 int power;
 if(m==1)
 power=x;
 else
 power=x*power1(x,m-1);
 return power;
}
```

3.3 Fibonacci series

```
int fib(int n)
{
 int fib1;
 if(n==0 || n==1)
 fib1=n;
 else
 fib1=fib(n-1)+fib(n-2);
 return fib1;
}
```

3.4 Recursion Properties

A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have −

- **Base criteria** − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- **Progressive approach** − the recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

3.4.1Time Complexity

In case of iterations, we take number of iterations to count the time complexity. Likewise, in case of recursion, assuming everything is constant, we try to figure out the number of times a recursive call is being made. A call made to a function is O(1), hence the (n) number of times a recursive call is made makes the recursive function O(n).

3.4.2 Space Complexity

Space complexity is counted as what amount of extra space is required for a module to execute. In case of iterations, the compiler hardly requires any extra space. The compiler keeps updating the values of variables used in the iterations. But in case of recursion, the system needs to store activation record each time a recursive call is made. Hence, it is considered that space complexity of recursive function may go higher than that of a function with iteration.