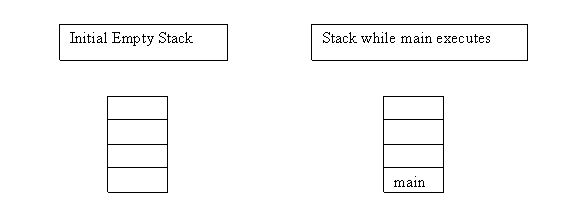
Recursion

When I first encountered recursion I thought: “This is simple, a function that calls itself.”  Naturally, I was soon confused and wondering what hit me - I had a new appreciation of the difficulties inherent in recursive processes.

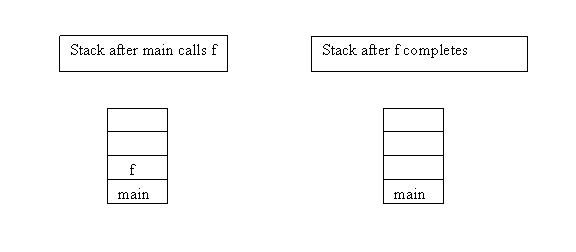
Let me explain the program stack and then show how that applies to recursion...

Every executable’s main is loaded into a program stack at the beginning of execution.  It remains there until it completes, at which time it is popped off of the stack.

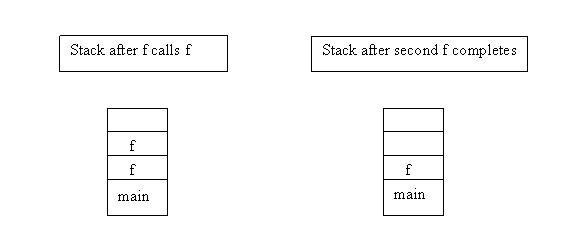


I will not go into all the details of what is in the stack as this will just cloud the discussion.

If main calls a function, that function is loaded onto the top of the stack and remains there until it is complete at which point it is popped off of the stack.



Now, a recursive function calls itself.  That means another instance of the function will be placed on the stack and will remain there until the function completes.



You need to look at a recursive function in terms of the program stack.  Lets use factorial as an example.  5 factorial is 5x4x3x2x1 = 120 and this can be implemented recursively.

int f(int x){

   if(x == 1) return 1;// line 1

   return f(x-1)\*x;    // line 2

}

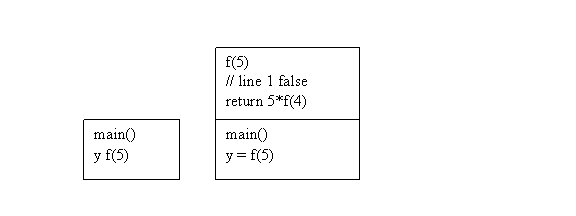
void main(){

   int y = f(5);// main call

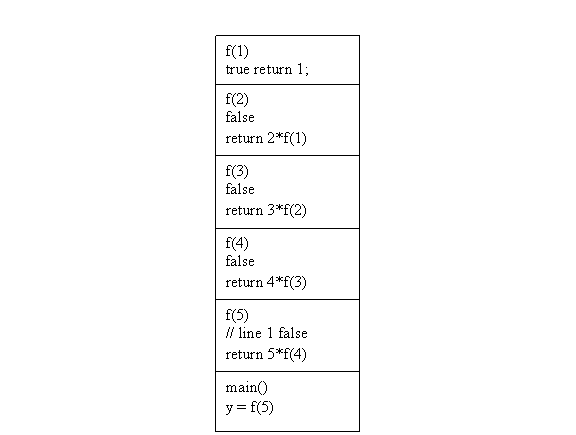
   // y should get 120

}

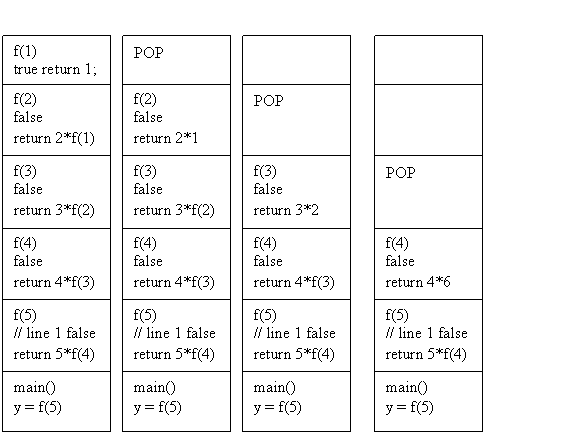
So lets watch the stack and see what happens.

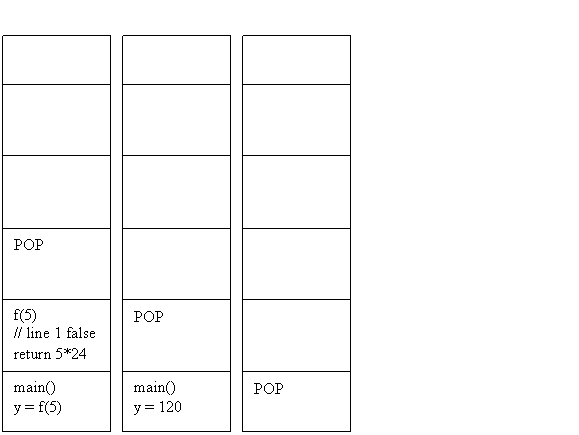


main calls the function f with a value of 5, so on the stack we get f(5).  Line 1 is false so we go to line 2 which calls f(4), etc.  Note that f(4) must complete before f(5) can complete.  f(5) will complete by returning 5\*f(4).  The stack will look like:



So at this point none of the functions have yet returned!  The first to return will be f(1) which will return 1.  Then f(2) will return 2. Then f(3) will return 6.  As in:





You can think of even the most complex recursive functions in terms of the program stack and you will then totally understand exactly what is going on in your program.

Admittedly, The more complex the function the more book keeping you will have to do.

Code for inorder traversal without recursion

void inOrder(struct Node \*root)

{

    stack<Node \*> s;

    Node \*curr = root;

    while (curr != NULL || s.empty() == false)

    {

        while (curr !=  NULL)

        {

            s.push(curr);

            curr = curr->left;

        }

          curr = s.top();

        s.pop();

        cout << curr->data << " ";

        curr = curr->right;

    }

}

Code for inoder traversal with recursion

void inOrder(struct Node \*root)

{

inOrder(root->left);

printf(“%d”,root->data);

inOrder(root->right);

}

There are some languages without loops such as Haskell and erlang whicjh does not have any looping functions but have very excellent applications are developed using those languages

Hence any further languages in developing stage there is a possibility that come without loops but

No language exist without recursion because recursion is a mathematical logic which can be newly invented or discovered by the user for his purpose on the given task

In python there occurs memory error when we iterate through large numbers so there is a replacement xrange instead of range so there is a chance that they would remove looping statements we can use recursion in replacement of that and so