Chapter 19: Recursion

Introduction

- A recursive algorithm is one that is **defined in terms of itself**. This means that, when the steps of the algorithm are written down, one of the steps refers to the same algorithm by name. When implemented, the subroutine will include one or more calls to the original algorithm.
- Recursion is a method of solving a problem where the solution depends on solving increasingly smaller instances of the same problem.

A basic example of Recursion

- Consider the problem of calculating the sum of all natural numbers from 1 to n, where n is the upper bound, or top, of the range. Natural numbers are unsigned whole numbers, sometimes they are called 'counting numbers'. So, the sum of all natural numbers from 1 to 5 is 15, because 1, plus, 2, plus, 3, plus, 4, plus, 5, equals, 15,1+2+3+4+5=15.
- The following example shows the subroutine sum_to_n expressed in pseudocode. You can see that, in the else clause, the subroutine sum_to_n is called again with the argument n-1. This **call to itself** is how you can identify that the subroutine is recursive.

```
1  FUNCTION sum_to_n(n)
2      IF n == 1 THEN
3          RETURN 1
4      ELSE
5          RETURN n + sum_to_n(n-1)
6      ENDIF
7  ENDFUNCTION
```

- The subroutine is defined with a single parameter, n. This value will be used as the upper bound of the range. If the subroutine is called with the argument 5, sum_to_n(5), the output would be 15. This is because 1, plus, 2, plus, 3, plus, 4, plus, 5, equals, 15,1+2+3+4+5=15

Question:
On which line is the recursive call made?

```
1  FUNCTION gcd(x, y)
2      IF y == 0 THEN
3          RETURN x
4      ELSE
5          new_y = x MOD y
6          new_x = y
7          RETURN gcd(new_x, new_y)
8      ENDIF
9  ENDFUNCTION
```

How Recursion works

- To understand how recursion works, let's return to the problem to calculate the sum of all natural numbers from 1 to n (where n is the upper bound).
- Let's start with giving n the value 5. You want to find out the sum of all natural numbers from 1 to 5. Apply the general rule:

  The sum of all natural numbers from 1 to 5 is:
  5 + the sum of all natural numbers from 1,1 to 4

- Now you must find out the sum of all natural numbers from 1 to 4. Again, you can apply the general rule:

  The sum of all natural numbers from 1 to 4 is:
  4 + the sum of all natural numbers from 1,1 to 3

- Now you must find out the sum of all natural numbers from 1 to 3. Again, you can apply the general rule:

  The sum of all natural numbers from 1 to 3 is:
  3 + the sum of all natural numbers from 1,1 to 2

- Now you must find out the sum of all natural numbers from 1 to 2. Again, you can apply the general rule:

  The sum of all natural numbers from 1 to 2 is:
  2 + the sum of all natural numbers from 1,1 to 1

- The next step is especially interesting. You have reached the **base case**. This is the point at which you know the answer to this version of the problem. The sum of all natural numbers from '1 to 1' is 1. The base case provides a **stopping condition**: the size of the problem is as small as it can get and the answer is known.

  The sum of all natural numbers from 1 to 1 is 1

- The stopping condition that is used for terminating a loop or a recursive algorithm is also known as a **rogue value** or **sentinel value**. In this example, the sentinel value is 1.
- So far, you have approached the problem by restating the general case over and over again until the base case is reached. Now you must **unwind** the algorithm to find the answer to the initial problem. If you work your way back through the stages, you can now insert the missing information by taking the answer from the previous stage.

Problem	Working out	Answer
The sum of all natural numbers from '1 to 1'		1
The sum of all natural numbers from '1 to 2'	$2 + 1$	3
The sum of all natural numbers from '1 to 3'	$3 + 3$	6
The sum of all natural numbers from '1 to 4'	$4 + 6$	10
The sum of all natural numbers from '1 to 5'	$5 + 10$	15

Thus, the sum of all natural numbers from 1 to 5 is 15.

Essential Characteristics of a Recursive Algorithm

- Every recursive algorithm must eventually stop calling itself; otherwise it will only stop when it has used up all of the available memory, which is known as **stack overflow**.
- When you write a recursive algorithm, you must use the following criteria:
  - Define a base case
  - Define the general case, referencing a version of the problem that decreases in size
  - When the algorithm is run, the base case will always be reached

Question:

- Which line expresses the base case?

```
1 FUNCTION print_backwards(string)
2     IF LEN(string) == 1 THEN
3         PRINT(string)
4     ELSE
5         print_backwards(string[1:])
6         PRINT(string[0])
7     ENDIF
8 ENDFUNCTION
```

Writing a Recursive Algorithm

- Having worked out the general case and the base case, you can now write code for the subroutine that will find the sum of all natural numbers from 1 to n.
- The **general case** for the function is:
  sum_to_n(n) = n + sum_to_n(*n*-1)
- The **base case** for the function is:
  sum_to_n(1) = 1

```
FUNCTION sum_to_n(n)
    IF n == 1 THEN
        RETURN 1
    ELSE
        RETURN n + sum_to_n(n-1)
    ENDIF
ENDFUNCTION
```

The subroutine has a single parameter, n.

- The if statement checks whether the **base case** has been reached. If so, the value 1 is returned.
- Otherwise, the subroutine is called again; this line uses the **general case**. The value that is passed (when the subroutine is called again) is reduced by 1 every time the subroutine is called. It will eventually be called with the value 1; this allows the base case to be reached and no further recursive calls will be made.

Parameters and return values in Recursive subroutines

Parameters

- In recursive subroutines, parameters that get their values from local variables within the recursive subroutine must be passed **by value**. If the parameter value is passed **by reference**, the subroutine will not produce the correct result. When you learn how to trace a recursive algorithm, you will see that it is important that each active copy of the subroutine has its own **local** variables; it should not have access to a shared memory location.
- The value that is passed into the subroutine must get smaller every time the subroutine is called. Eventually, it must allow the base case to be reached. If the base case is not reached, the recursion will not stop. Or at least, it will not stop in theory. In practice, the subroutine will continue to run until there is a **stack overflow**

Return values

- When you code a return statement, you are instructing the translator that, at this point, the subroutine should terminate and return a value. Once a return statement is run, no more lines of that subroutine will be executed.
- In non-recursive subroutines, it is generally good practice to code only one return statement on the last line of the subroutine. This makes it clear to anyone reading your code what value, if any, is returned. However, recursive subroutines will often, correctly, have multiple return statements; this feature allows the subroutine to handle both base and general cases.

Tracing a Recursive Algorithm

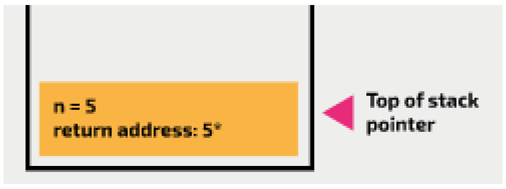
- Whenever a subroutine is called, an entry is made on the **call stack** in the form of a **stack frame**. In simple terms, this is a record that contains:
    - The value of any parameters and local variables
    - The position in the subroutine that was reached when another subroutine was called
- To trace a recursive subroutine, you may find it useful to sketch out the stack frames. Let's return to the recursive subroutine to calculate the sum of all natural numbers from 1 to n,*n*. This is shown in pseudocode only. The lines have been numbered so that they can be cross referenced.

```
1 FUNCTION sum_to_n(n)
2     IF n == 1 THEN
3         RETURN 1
4     ELSE
5         RETURN n + sum_to_n(n-1)
6     ENDIF
7 ENDFUNCTION
```

- Notice that the recursive call is made **halfway through line 5**. This means that the subroutine is called again before the rest of the line is executed. In the example below, the return address in the stack frame is shown as **5***.
- You have already learned that when you run the subroutine with the value of 5 as the argument, you will get the answer 15 (5, plus, 4, plus, 3, plus, 2, plus, 1, equals, 15,5+4+3+2+1=15). You can follow this step by step, as shown below.

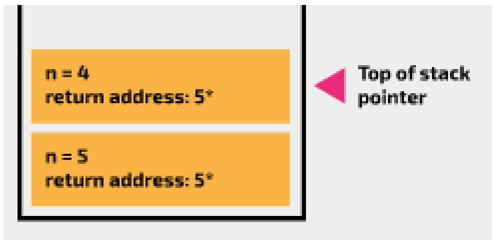
Step 1

A stack frame for the first call of the subroutine is pushed on to the call stack. This includes the value for n,*n*, which is 5. The subroutine runs to line 5 and then calls itself with the argument value of n, minus, 1,*n*−1, which in this case is 4. The return address of 5* is updated on the stack frame.



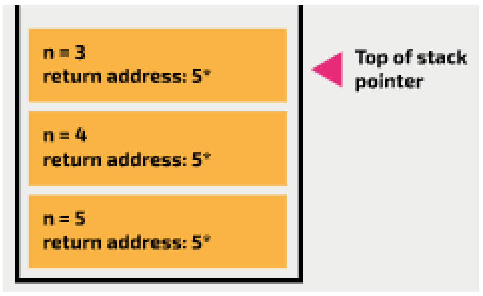
Step 2

A stack frame for the second call of the subroutine is pushed on to the call stack. This includes the value for n,*n*, which is 4. The top of stack pointer is incremented by 1. The subroutine runs to line 5 and then calls itself with the argument value 3. The return address of 5* is updated on the current stack frame.



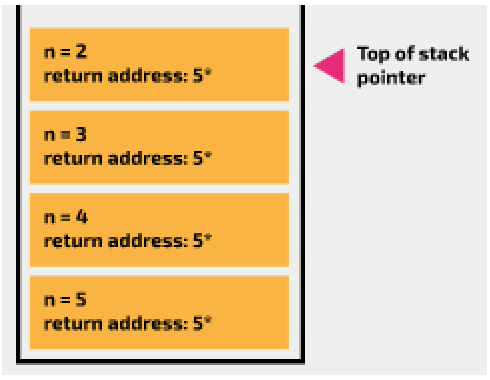
Step 3

A stack frame for the third call of the subroutine is pushed on to the call stack. This includes the value for n,*n*, which is 3. The top of stack pointer is incremented by 1. The subroutine runs to line 5 and then calls itself with the argument value 2. The return address of 5* is updated on the current stack frame.



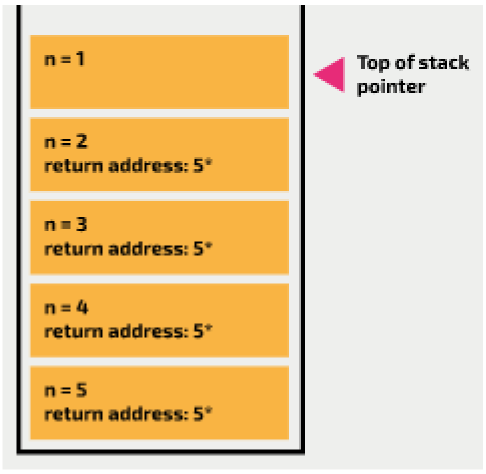
Step 4

A stack frame for the fourth call of the subroutine is pushed on to the call stack. This includes the value for n,*n*, which is 2. The top of stack pointer is incremented by 1. The subroutine runs to line 5 and then calls itself with the argument value 1. The return address of 5* is updated on the current stack frame.



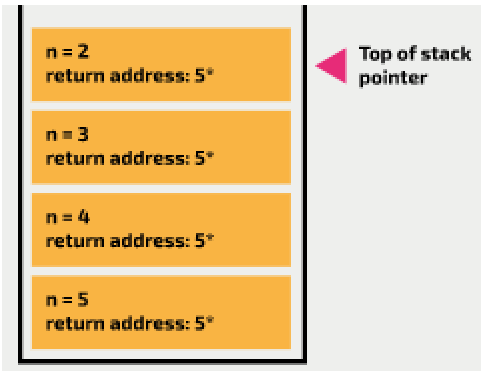
Step 5

A stack frame for the fifth call of the subroutine is pushed on to the call stack. This includes the value for n,*n*, which is 1. The top of stack pointer is incremented by 1.

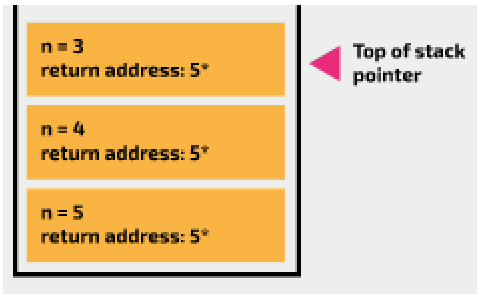


Step 6

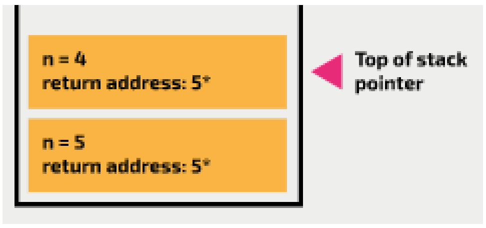
The subroutine checks the condition at line 2. This evaluates as 'True', so the subroutine returns 1. The return statement causes the subroutine to terminate. The frame is popped from the call stack and the top of stack pointer is decremented by 1.



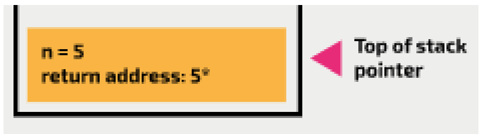
Step 7
Control returns to the subroutine at the top of the stack and it goes back to line 5 (remember — **halfway through line 5**). This line can now complete by taking the value of n,*n*, which is 2, from the stack frame and adding 1 (the value that was returned at the previous step). The result — 3 — is returned. The return statement causes the subroutine to terminate; the frame is popped from the call stack and the top of stack pointer is decremented by 1.



Step 8
Control returns to the subroutine at the top of the stack and it goes back to line 5. This line can now complete by taking the value of n,*n*, which is 3, from the stack frame and adding 3 (the value that was returned at the previous step). The result — 6 — is returned. The return statement causes the subroutine to terminate. The frame is popped from the call stack and the top of stack pointer is decremented by 1.



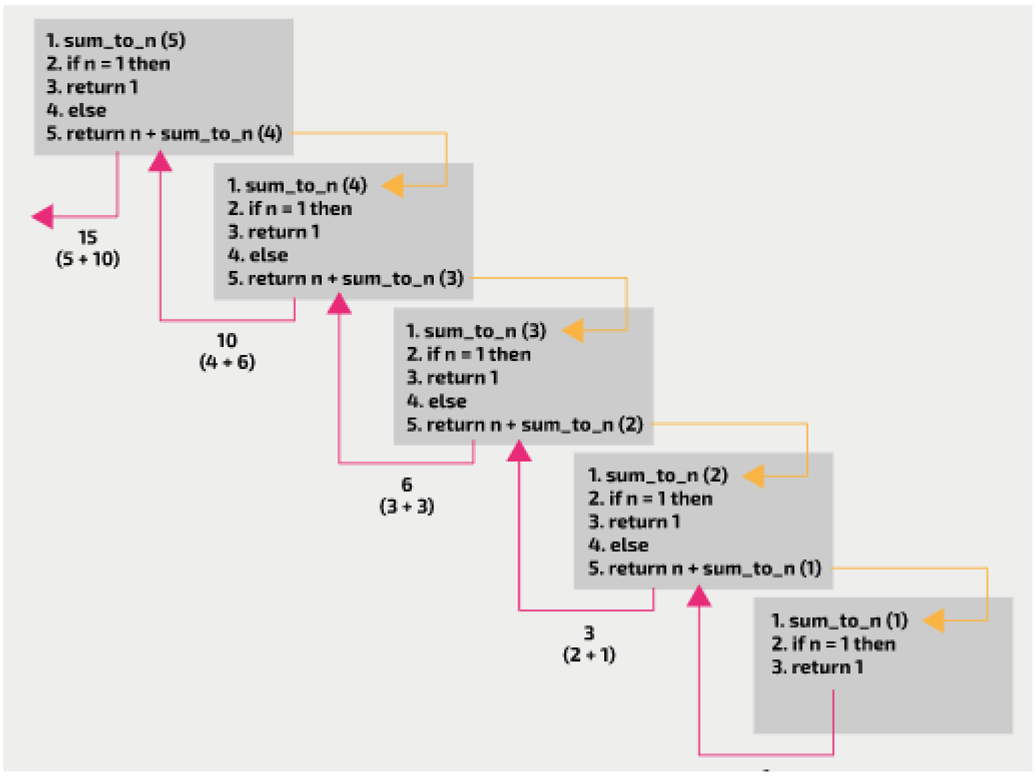
Step 9
Control returns to the subroutine at the top of the stack and it goes back to line 5. This line can now complete by taking the value of n,*n*, which is 4, from the stack frame and adding 6 (the value that was returned at the previous step). The result — 10 — is returned. The return statement causes the subroutine to terminate. The frame is popped from the call stack and the top of stack pointer is decremented by 1.



Step 10
Control returns to the subroutine at the top of the stack and it goes back to line 5. This line can now complete by taking the value of n,*n*, which is 5, from the stack frame and adding 10 (the value that was returned at the previous step). The result — 15 — is returned. The return statement causes the subroutine to terminate. The frame is popped from the call stack and the top of stack pointer is decremented by 1.

The stack is now empty and the program terminates.

Before you move on, make sure that you can follow this step-by-step trace with confidence. The diagram below shows another way that the flow of calls and returns can be represented. If you have to trace a recursive subroutine, you may find one (or both) of these methods useful.



Trace tables

- When you trace an algorithm, you are often asked to present your results as a **trace table**. Here is an example of a completed trace table for the same algorithm. Notice how, when call number 5 is handled, program flow returns to the still open call 4, call 3, call 2, and finally call 1. It only returns a value when it completes line 5 of the code.

Call number	n	n equals 1?	Return value
1	5	False	
2	4	False	
3	3	False	
4	2	False	
5	1	True	1
4	2		3
3	3		6
2	4		10
1	5		15

Question:

The recursive subroutine described in pseudocode below prints a string backwards.

```
1 FUNCTION print_backwards(string)
2     IF LEN(string) == 1 THEN
3         PRINT(string)
4     ELSE
5         print_backwards(string[1:])
6         PRINT(string[0])
7     ENDIF
8 ENDFUNCTION
```

Trace the subroutine when called with the string "pizza".

To help you, the trace table shown below is partially complete. Copy the table out on a piece of paper and complete the values for the shaded cells. When you have finished, reveal the answer to check your work.

Trace the subroutine when called with the string "pizza".

To help you, the trace table shown below is partially complete. Copy the table out on a piece of paper and complete the values for the shaded cells. When you have finished, reveal the answer to check your work.

Function call/line	String	LEN(string)	string[1:]	printed
1 (line 1)	pizza	5	izza	
2 (line 1)	izza	🟧	🟧	
3 (line 1)	zza	🟧	🟧	
4 (line 1)	🟧	🟧	🟧	
5 (line 1)	🟧	🟧	🟧	🟧
4 (line 6)	za			🟧
3 (line 6)	zza			🟧
2 (line 6)	🟧			🟧
1 (line 6)	🟧			🟧

Function call/line	String	LEN(string)	string[1:]	printed
1 (line 1)	pizza	5	izza	
2 (line 1)	izza	4	zza	
3 (line 1)	zza	3	za	
4 (line 1)	za	2	a	
5 (line 1)	a	1		a
4 (line 6)	za			z
3 (line 6)	zza			z
2 (line 6)	izza			i
1 (line 6)	pizza			p

An iterative alternative to Recursion

- For every recursive algorithm, there is always an iterative solution (the same problem can be solved using loops). Consider an alternative solution that uses iteration to solve the problem of calculating the sum of all natural numbers from 1 to n,*n*

```
FUNCTION sum_to_n_iterative(n)
    total = 0
    FOR i = 1 TO n
        total = total + i
    NEXT i
    RETURN total
ENDFUNCTION
```

This time there is only a single function call. A loop is used to handle the successive values of n,*n*.

Why use Recursion?

Recall that for every recursive solution, there is an iterative solution (the same problem can be solved using loops). You may find recursion difficult and wonder why you should use it.

One of the main reasons that recursion is used is because many data structures are, by their nature, recursive. This means that it is more intuitive to process them recursively. A good example of this is the traversal algorithm for a binary tree; you will see that the recursive solution has simple, elegant code.

Recursive algorithms tend to have fewer steps than non-recursive solutions. This means that they are easier for humans to read. However, they are not necessarily very easy to trace or debug, as you need to be familiar with the call stack and how it is used. Having multiple concurrent versions of the same procedure can be confusing.

Recursive algorithms use more memory. Because a stack frame has to be added to the stack with each recursive call, and values of any local variables (including values passed in for parameters) need to be stored until the stack frame is removed from the call stack, the memory allocation is greater than that of an iterative function. For the same reasons, recursion can be slow; time needs to be spent on stack operations to support recursion.

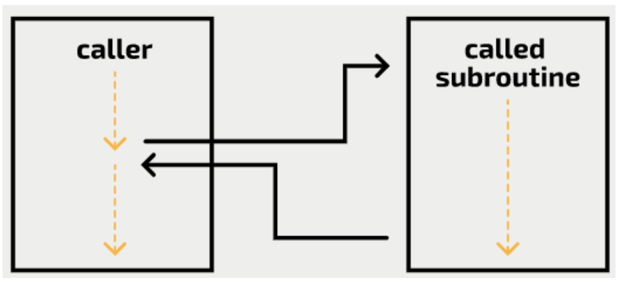
Arguments for recursion	**Arguments against recursion**
Natural way to process data structures, such as trees, that are recursive by nature.	
Fewer lines of code mean that the function is easier to read.	Harder to trace because you must keep track of multiple instances of the function.
	Uses more memory because of the need to store multiple stack frames.
	Slower because of the need to manage stack operations.

The Call Stack and Stack Frames

- When a subroutine is called, an entry is placed on the **call stack**.
- If you have already studied the 'Data structures' topic, you should recall that a stack is a LIFO (last in, first out) data structure. If you have not studied the 'Data structures' topic, take the time to do so before you study this concept.

What happens when a subroutine is called?

- When a subroutine is called from another subroutine (the caller), several things must happen:
    - The parameter values must be passed from the caller to the called subroutine
    - The state of the caller must be saved before program control is switched to the called subroutine
    - The called subroutine must be executed
    - Return value(s) from the called subroutine must be passed back to the caller
    - The state of the caller must be reinstated so that it can finish executing at the point immediately following the position that the subroutine call was made



Question:

```
PROCEDURE main()
    user_response = INPUT("Please enter a number > ")
    number = FLOAT(user_response)
    answer = squared(number)
    PRINT(number + " squared is " + answer)
ENDPROCEDURE

FUNCTION squared(n)
    n_squared = n * n
    RETURN n_squared
ENDFUNCTION
```

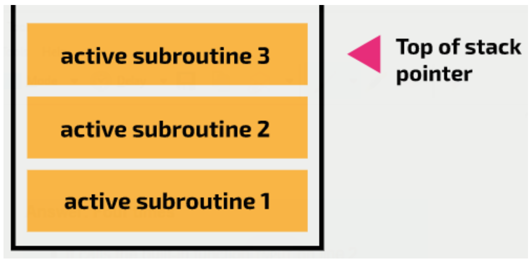
- When main is run, it will not complete execution until it reaches the end of the procedure. In the pseudocode example of the program, how many times does main call other subroutines?

Answer: 4 times

- It calls the built-in function INPUT on line 2
- It calls the built-in function FLOAT on line 3
- It calls the user-defined function squared on line 4
- It calls the built-in function PRINT on line 5

The Call Stack

- A call stack is an area of memory that is allocated to a program to store information about each **active subroutine**. An active subroutine is one that has been called, but has not yet finished execution. At any point during a program's execution, there may be multiple active subroutines.



- The call stack, like any stack, is finite in size. Memory space is allocated for the call stack before the program is run. In some low-level languages, the amount of memory allocated is determined by the programmer. In high-level languages, it is controlled by the program language variables, the compiler, and/or the operating system.
- If you write a subroutine that calls itself with no means of escape, you will fill up the stack with hundreds of stack frames. This will eventually result in the infamous **stack overflow** error message.

Stack Frames

- When a subroutine is called, a **stack frame** is created with memory space allocation for:
  - The values of any local variables
  - The values of any parameters
- It also holds a reference to the 'execution point' of the subroutine; think of this as the line number where program control needs to return to if another subroutine is called. Sometimes this is called the **return pointer**.
- To see how the call stack and stack frames are used, consider this program that is made up of two subroutines.

```
1  PROCEDURE main()
2      num1 = INPUT("Enter a number ")
3      num2 = INPUT("Enter another number ")
4      calculate_product(num1, num2)
5      PRINT("I hope you enjoyed using the product calculator")
6  ENDPROCEDURE
7
8  PROCEDURE calculate_product(n1, n2)
9      product = n1 * n2
10     PRINT(product)
11 ENDPROCEDURE
```

- When the main subroutine is called, a stack frame is created on the call stack. This will have memory space for the local variables num1 and num2, and space to store the **return pointer**. The return pointer is the address of the line within the subroutine that will be run next.
- **Note:** It is important to understand that one line of high-level code will result in many lines of low-level code. The return pointer will actually point to the memory location of a low-level instruction, but for illustration purposes we will show the return pointer pointing to a line in the code.
- Now, the subroutine main is running:
    - **Line 7**: the user is prompted to enter a number. The value entered is stored in the memory reserved within the stack frame for the variable num1.
    - **Line 8**: the user is prompted to enter another number. The value entered is stored in the memory reserved within the stack frame for the variable num2.
    - **Line 9**: the subroutine calculate_product is called. The stack frame for main holds the return pointer for line 10, as this will be the next line of main to be executed.
- The example stack frame shown below assumes that the value 12 was entered for num1 and the value 6 was entered for num2.

num1: 12
num2: 6

return pointer: 10

- At this point, another stack frame is created (on the call stack) for the subroutine calculate_product. It will have memory allocation for its parameters (n1 and n2) and for the local variable product, as well as space to store the return pointer. The **values** of num1 and num2 that are passed into the subroutine (from main) are stored at the locations for n1 and n2.
- Now, the subroutine calculate_product is running:
    - **Line 2**: the two numbers are multiplied together and the result is stored in the memory reserved for the variable product.
    - **Line 3**: the value of product is displayed to the user.
- The subroutine calculate_product has completed. The associated stack frame is popped (removed) from the stack.
- A peek (look) operation is carried out and program execution continues with the frame at the top of the stack; the stack frame for main. This subroutine continues execution at the point indicated by the return pointer (line 10).
    - **Line 10**: the message "I hope you enjoyed using the product calculator" is displayed to the user.
- There are no more instructions, so the main subroutine has completed. The associated stack frame is popped (removed) from the stack.
- A peek (look) operation is carried out, but the call stack is empty so the program can terminate

Calling a subroutine as part of an expression

```
 1  PROCEDURE main()
 2      hours = INPUT("Enter hours worked")
 3      rate = INPUT("Enter hourly rate")
 4      pay = calculate_pay(hours, rate)
 5      PRINT("Your pay is " + pay)
 6  ENDPROCEDURE
 7
 8  FUNCTION calculate_pay(h, r)
 9      pay =  h * r
10      RETURN pay
11  ENDFUNCTION
```

Tracing the pseudocode version of the program

In this example, you will see that the subroutine calculate_pay is called on line 4. This line of code is an **expression**; in this case an assignment statement, where the value that is assigned to the variable named pay is obtained by calling and executing the subroutine calculate_pay.

This means that, when the subroutine calculate_pay is called, the specific line of code that makes the subroutine call (line 4) has not finished running. When the subroutine calculate_pay has completed, control must return to the exact point where the subroutine was called. The value returned by calculate_pay will be assigned to the variable pay.

Clearly, the return pointer cannot be set to 'halfway through line 4'. When the high-level code is translated to low-level instructions, line 4 is split into more than one instruction. The return address will be set to the next line of low-level code to be executed.

Return statements in subroutines

- When you code a return statement, you are instructing the translator that, at this point, the subroutine should terminate and return a value. Once a return statement is run, no more lines of that subroutine will be executed and the stack frame for that subroutine will be removed from the call stack.
- It is generally good practice to code only one return statement on the last line of the subroutine. This makes it clear to someone else who is reading your code what value, if any, is returned. Some programming languages, such as Python, allow multiple values to be returned from a single return statement. In other languages, you may have to use a tuple to return multiple values.
- When a subroutine is coded with multiple return statements, or return statements are hidden within dense lines of code, the subroutine may be hard to debug. Be especially careful when coding return statements within conditional structures, to make sure that your functions always return something.
- Recursive subroutines will often, correctly, have multiple return statements. This feature allows the subroutine to handle both base and general cases.

Program memory management

When a program has finished running, the memory allocated to the call stack is made available again. As a high-level programmer, you do not usually have to worry about any of this as it is handled for you.

There are other sections of memory that are allocated to programs. Most languages use a **heap** to store dynamic structures, such as objects and linked lists that are created at runtime. Typically, there is also a **data section** that is used for global variables.