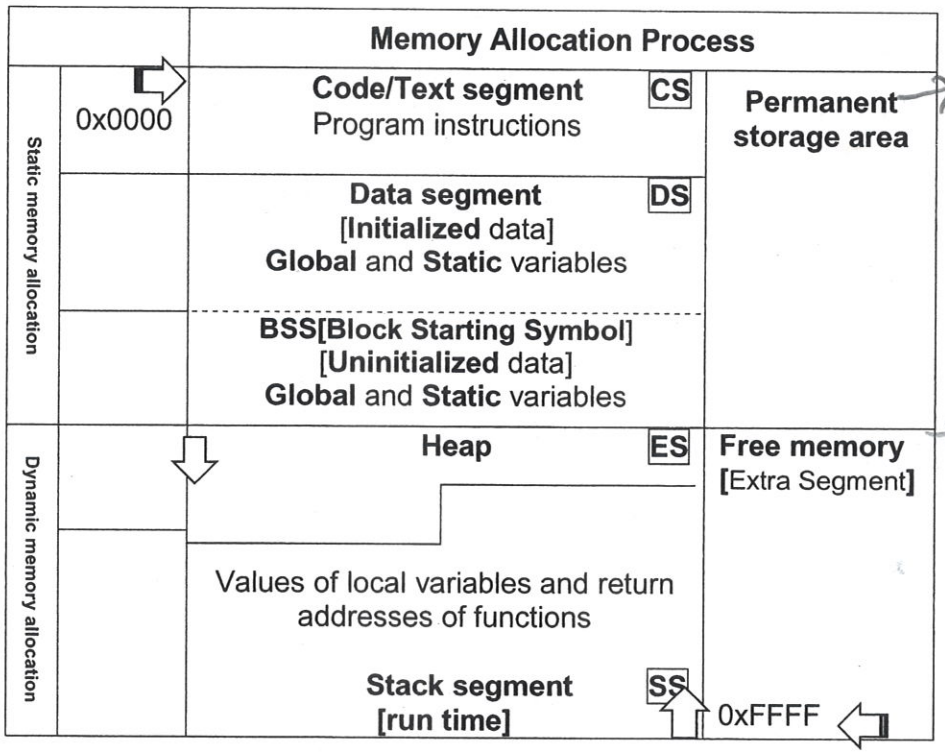
TEMASEK JUNIOR COLLEGE

Temasek Junior College

JC H2 Computing

Problem Solving & Algorithm Design 7-1 – Memory Allocation

1 Memory Allocation Process



The **code segment [CS],** also known as a text segment, is where the corresponding section of the program's address space that contains executable instructions is stored and is generally read-only and fixed size.

The ***data* segment [DS]** contains any global or static variables which have a *pre-defined value* and *can be modified*. That is any variables that are not defined within a function (and thus can be accessed from anywhere) or are defined in a function but are defined as *static* so they retain their address across subsequent calls.

The **BSS [Block Starting Symbol]** , also known as *uninitialized data*, is usually adjacent to the data segment. The BSS segment contains all global variables and static variables that are *initialized to zero or do not have explicit initialization* in source code.

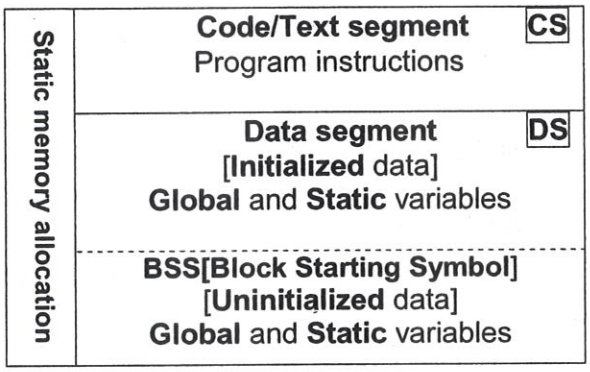
Global variables, **static** variables and **program instructions** get their memory in the permanent storage area whereas **local** variables and **return addresses** of called functions are stored in a memory area called **stack**.

The **heap area** [**ES** Extra Segment] commonly begins at the end of the bss and data segments and grows to larger addresses from there. **The size of heap keeps changing.** The heap area is shared by all threads, shared libraries, and dynamically loaded modules in a process. This region is used for **dynamic memory allocation** during execution of the program.

The **stack segment [SS]** contains the program stack, a LIFO (Last In First Out) structure, typically located in the higher parts of memory. A "stack pointer" register tracks the top of the stack; it is adjusted each time a value is "pushed" onto the stack. The set of values pushed for one function call is termed a **"stack frame"** or **"activation record"**. A stack frame consists at minimum of a ***return address***. Local variables are also allocated on the stack.
The stack area adjoined the heap area and they grew towards each other; when the stack pointer met the heap pointer, free memory was exhausted, then "**Memory overflow**" fatal error occurred.

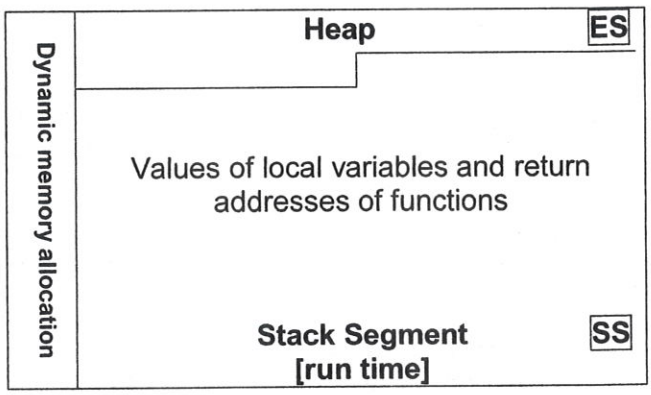
1.1 Static Memory Allocation

- Static memory allocation is an allocation procedure.
- In this type of allocation, allocation of data objects is done at compile time only.
- In object-oriented programming (OOP), static memory allocation is used for allocation of all the data objects at compile time.
- In static allocation, the compiler decides the extent of storage which cannot be change with time As such, it is easy for the compiler to know the addresses of these data objects in the activation records at later stage.
- **Program instructions, global** variables and **static** variables get their memory in the permanent storage area



2 Dynamic Memory Allocation

- Dynamic memory allocation is the process of assigning the memory space during the execution time or the runtime.
- Dynamic memory allocation can be done on both the stack and the heap.
- An example of dynamic allocation done on the stack is recursion where the functions are put into the call stack in order of their occurrence and popped off one by one on reaching the base case.



2.1 Heap Allocation

- The heap helps in managing the **dynamic memory allocation**.
- Heap allocation is an allocation procedure in which the heap is used to manage the allocation of memory.
- In heap allocation, creation of dynamic data objects and data structures is also possible as with stack allocation.
- Heap allocation overcomes the limitation of stack allocation.
- It is possible to retain the value of variables even after the activation record in the heap allocation strategy, which is not possible in stack allocation.
- A linked-list is maintained for the free blocks and deallocated space is reused with the best fit.
- While allocating memory on the heap, we need to delete the memory manually as memory is not freed (deallocated) by the compiler itself even if the scope of allocated memory finishes (as opposed to the case of stack).

2.2 Stack Allocation

- Stack allocation is a temporary memory allocation scheme. It is a procedure in which a stack is used to organize the storage.
- The stack used in stack allocation is known as the control stack.

Activation records or Stack frames

Return address	The function is called and data passed to it. The return address is placed on the stack so that when the function is finished, it will look at the return address so it knows where to go back to.
:	
Reture Value	
:	
Function call and argument data	The subroutine is running using local variables. When a function is called, the last position of the stack frame is saved as a saved frame pointer.
Saved frame pointer	

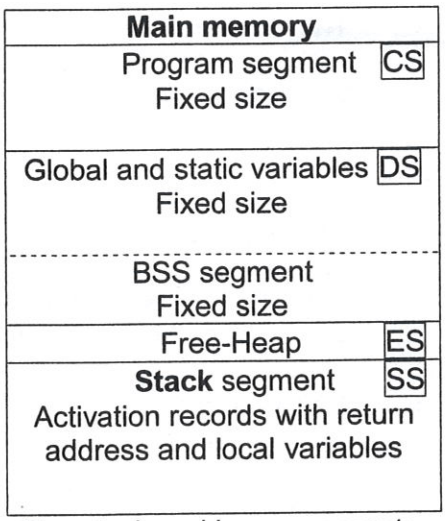
- In stack allocation, creation of data objects is performed dynamically.
- Stacks can be used to store information about a running program. In this case it is known as a **stack frame** and a pointer will be used to identify the start point of the frame. This is used as a **call stack** when running programs as it can be used when a subroutine is running to call functions and pass parameters and arguments.
- Activation records (Stack frames) are created for the allocation of memory and pushed into the stack using the Last in First out (LIFO) method
- **When a function is called, the local variables** and **return addresses** are stored in the **stack as activation record**.
- The corresponding error, **StackOverFlowError** is received, if the stack memory is filled completely before completion of the method.
- Stack memory allocation is considered safer as compared to heap memory allocation because the data stored can only be accessed by the owner thread.
- Stack-memory has less storage space as compared to Heap-memory.

3 Differences between Static and Dynamic Memory Allocation

No.	Static Memory Allocation	Dynamic Memory Allocation
1	Variables get allocated permanently.	Variables get allocated only when the program unit is active.
2	Memory is allocated at compile time.	Memory is allocated at runtime.
3	Occurs before program execution.	Occurs during program execution.
4	Uses stack for managing the static allocation of memory.	Uses heap for managing the dynamic allocation of memory.
5	Less efficient.	More efficient.
6	However, execution is faster than dynamic memory allocation.	However, execution is slower than static memory allocation.
7	No memory reuse allowed.	Memory reuse allowed and memory can be freed when not in use.
8	Once memory is allocated, memory size cannot be changed.	When memory is allocated, the memory size can be changed accordingly.
9	Memory which is unused cannot be reused.	Reusing of memory is allowed. User can allocate more memory when required. Also, user can release the memory when it is needed for further uses.
10	Allocated memory remains from start to end of the program.	Allocated memory can be released at any time during the program.
11	Example: Used for array data structure	Example: Used for linked-list data structure

4 Scope of Function in Main Memory during Runtime

- When **new module** (procedure, function or method) is called (the stack takes the role for the process)
  - the caller is suspended
  - the "state" of caller is saved
  - new space is allocated for variables of the new module
- At the end of new **module**
  - release the space allocation.
  - return to the point next to the caller with the previous "state" recovered.



The stack and heap segments, on the other hand, grow and shrink during program execution.

Example 1

Below is an algorithm that uses *global* and *local* variables.

(a) Write down all *global* and *local variable* names in the boxes provided from the algorithm above.

(b) Explain the difference between *global* and *local* variables.

```
Num1 is integer  {number input by user}
Num2 is integer  {number input by user}

function DivNums(n1, n2)  //Parameters
n1, n2
    Num1 ← n1+n2
    Num2 ← n1-n2
    n1 ← Num1
    n2 ← Num2
    return n1/n2
endfunction

procedure AddTwoNum() {procedure to find the total of two integers}
    Total is integer        //
    Total ← Num1 + Num2
    Print Total
endprocedure

function SubTwoNum(m1, m2)  //
    Num1 is integer          //
    Num2 is integer          //
    Difference is integer    //
    Num1 ← m1 * 2
    Num2 ← m2 / 2
    Difference ← Num1 - Num2
    return Difference
endfunction

main
output "type in first number"
    input Num1
    output "type in second number"
    input Num2
    CALL AddTwoNum()
    Print DivNums(Num1, Num2)
    Print SubTwoNum(Num1, Num2)
    CALL AddTwoNum()
    Print SubTwoNum(n1, n2) //
endmain
```

Module-related	Variable Name	Global or Local
Current project	Num1	G
	Num2	G

	n1	L
	n2	L
	Num2	G
	Num1	G

	m1	L
	m2	L
	Num1	G
	Num2	G

5 Calling Procedures or Functions via a Stack

Example 2

Memory Allocation Process

Static memory allocation [permanent while program is running]

CS

Program instructions

```
procedure AddTwoNum()
      Total is integer
      Total ← Num1 + Num2
      Print Total
endprocedure

function SubTwoNum(m1, m2)
      Num1 is integer
      Num2 is integer
      Difference is integer
      Num1 ← m1 * 2
      Num2 ← m2 / 2
      Difference ← Num1 - Num2
      return Difference
endfunction

function DivNums(n1, n2)
      Num1 ← n1+n2
      Num2 ← n1-n2
      n1 ← Num1
      n2 ← Num2
      return n1/n2
endfunction

main
      output "type in first number"
      input Num1
      output "type in second number"
      input Num2
      CALL AddTwoNum()
      Print DivNums(Num1, Num2)
      Print SubTwoNum(Num1, Num2)
      CALL AddTwoNum()
      Print SubTwoNum(n1, n2)
endmain
```

DS

Global variables table		
Identifier	**Type**	**Pointer**
Num1	integer	10A1
Num2	….	10A9

Static variables table		
Identifier	**Type**	**Pointer**

Literals table		
…	"type in first number"	…
…	"type in second number"	…

BSS

ES **Heap** Free for memory, local variables and return addresses for functions

Dynamic memory allocation

1A01	1A05	10A9	← Memory address of each 4 bytes					

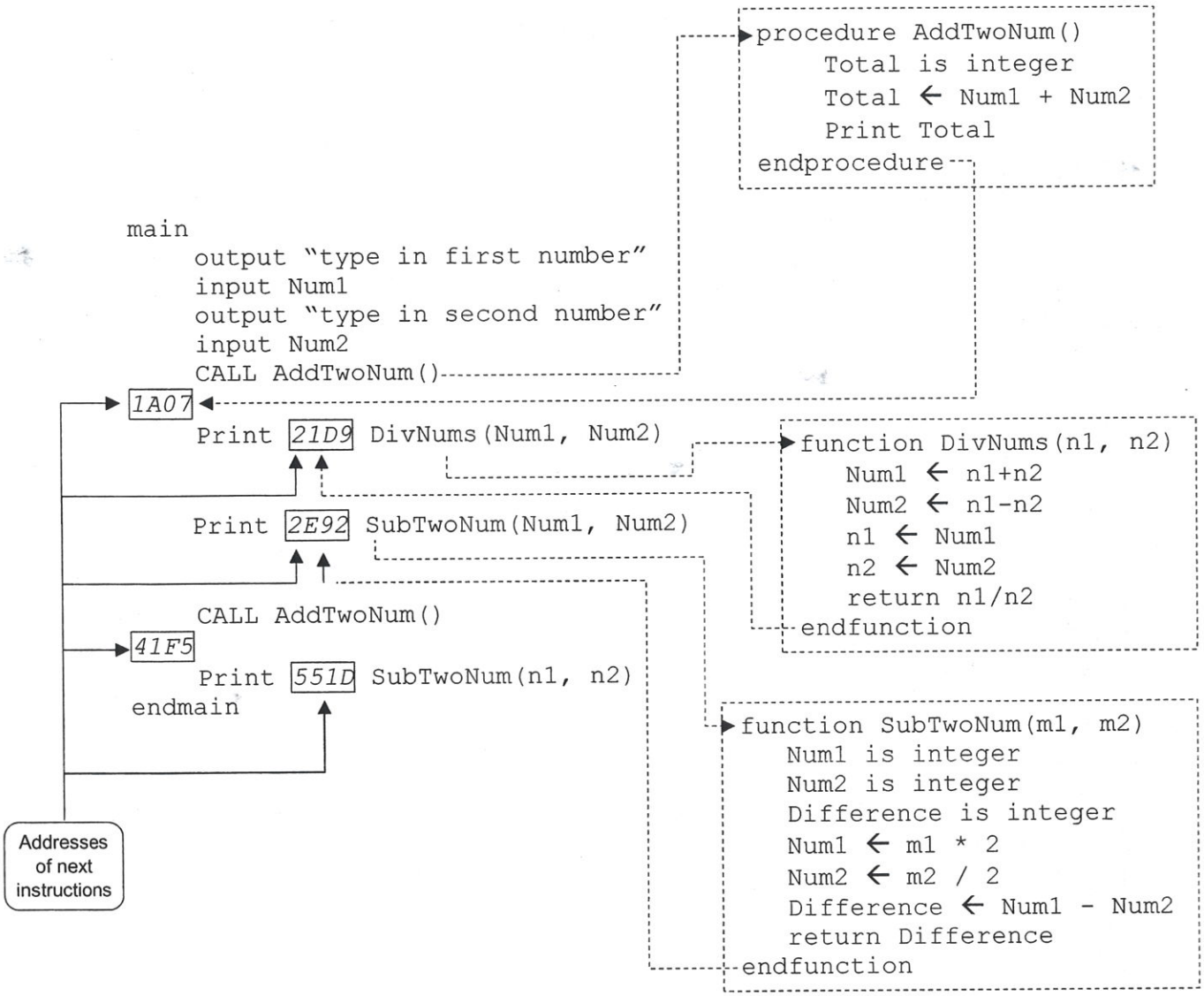
SS

Module-related	Identifier	Classification	Type	Pointer
DivNums		function	Real	
DivNums	n1	parameter	integer	
DivNums	n2	parameter	integer	
AddTwoNum		procedure	….	
SubTwoNum		function	….	
SubTwoNum	m1	parameter	….	
SubTwoNum	m2	parameter	….	
main		function	….	

Stack [run time]

- When a procedure or function is called, the computer needs to know where to return to when the function or procedure call is completed. Hence the return address must be stored.
- Also, functions and procedures may call other functions and procedures, which means that not only must several return addresses be stored in the correct order, they must be retrieved in the correct order.
- This is achieved by using a stack.
- The figure below shows what happens when three functions are called, one after another. The numbers represent the addresses of the instructions following the call to each function. The return addresses have to be stored and retrieved in the correct order.

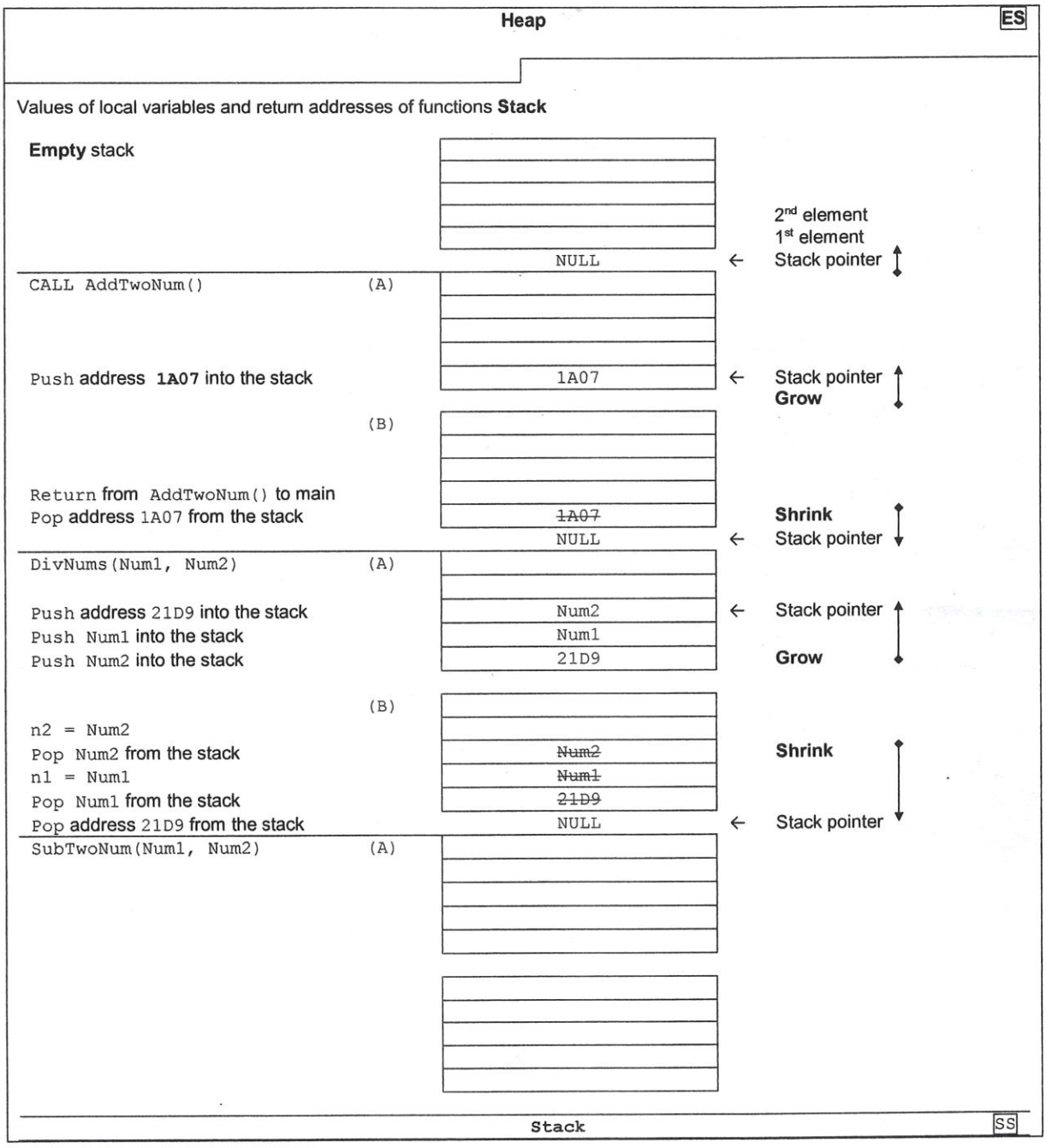
- The return addresses to be stored by this program execution are: 1A07, 21D9, 2E92, 41F5 and 551D, in that order.
- They need to be retrieved in the order 1A07 then 21D9, 2E92, 41F5 and finally 551D.



6 Using a Stack to Pass Parameters

Now suppose that values need to be passed to or from a function or procedure. A stack can be used.

Consider a main program that calls three modules.



The last address stored is the first to be retrieved, i.e. it is a stack.

Notice that when the values are read (popped off the stack), they remain in the stack, it is just that they can never be used again and at some point the value is overwritten. They can be considered as deleted once they have been popped.

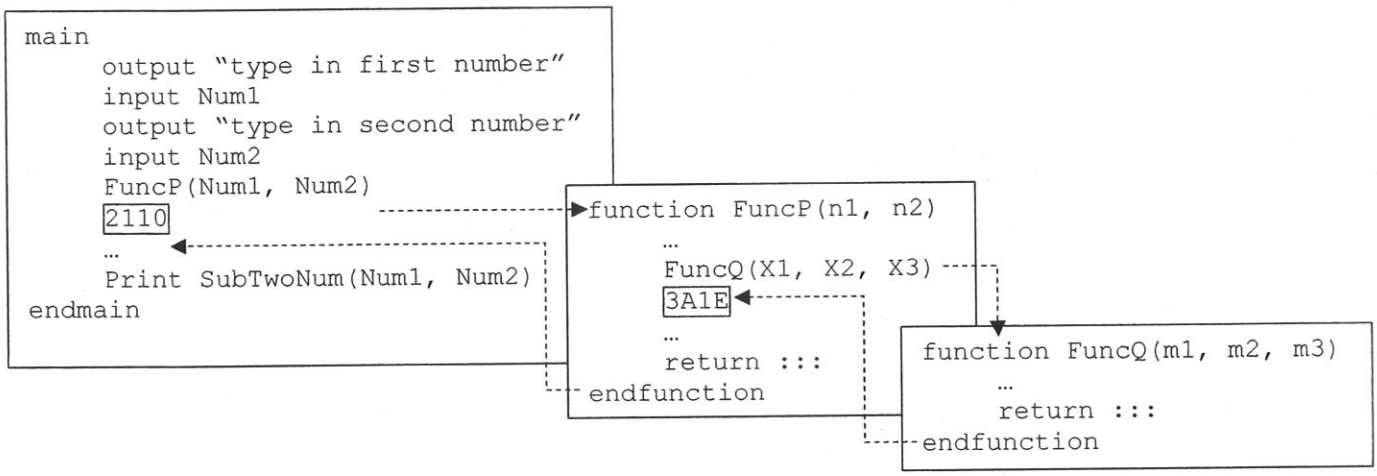
7 Using a Stack to Monitor Linear Calling of Functions

Example 3
Consider a main program that calls two functions:

```
FuncP(n1, n2)
FuncQ(m1, m2, m3)
```

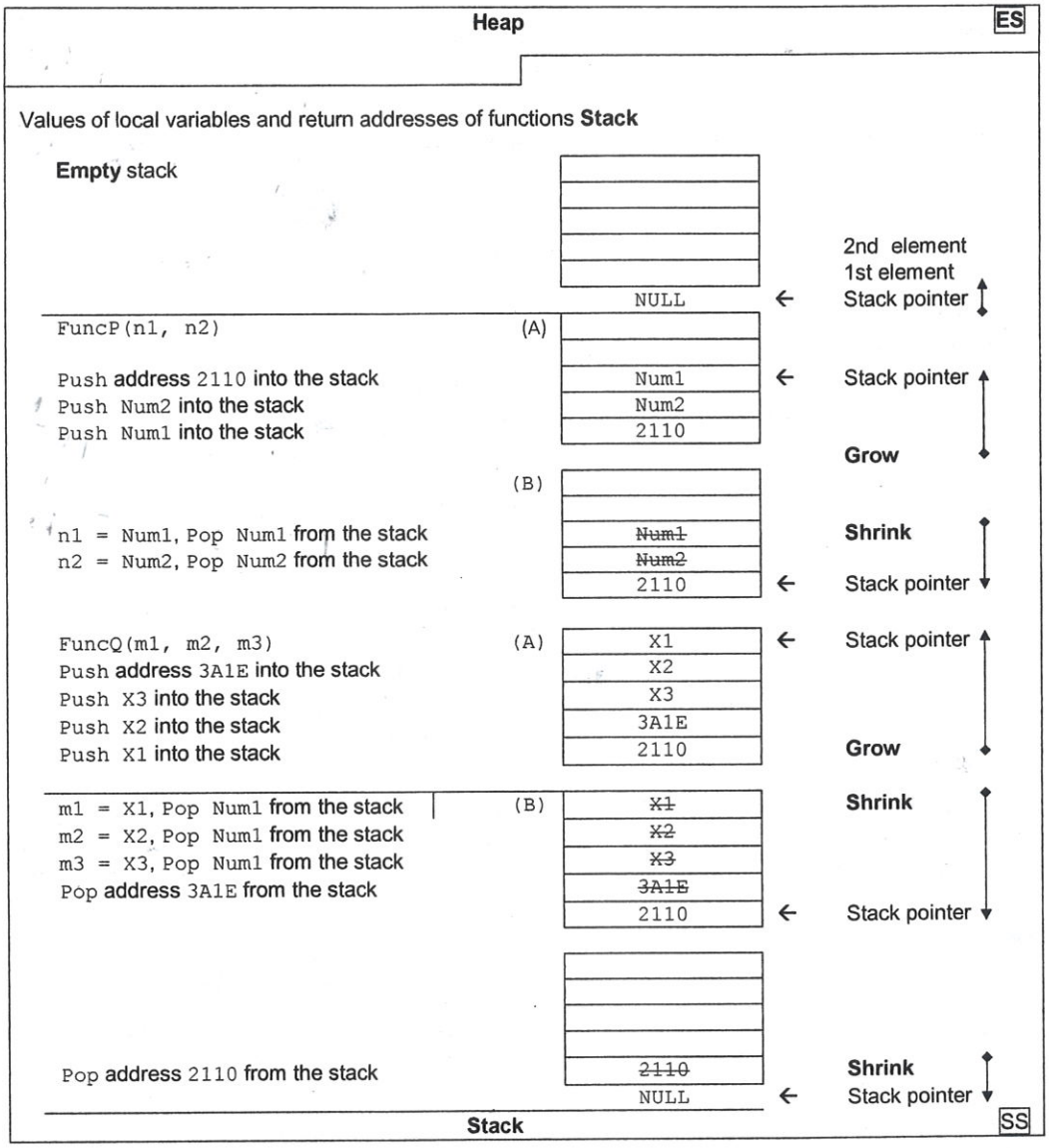
`n1` and `n2` are the formal parameters for `FuncP`.
`m1`, `m2` and `m3` are the formal parameters for `FuncQ`.

`Num1` and `Num2` are arguments used when calling `FuncP(Num1, Num2)`
`X1`, `X2` and `X3` are arguments used when calling `FuncQ(X1, X2, X3)`



The figure in the next page shows how the procedures are being called and the return addresses that must be placed on the stack.

- In interpreting the diagram, let us suppose that all the parameters are passed by value.
- When a procedure is called, the actual parameters are placed on the stack. The procedure pops the values off the stack and stores the values in the formal parameters.
- Notice how the stack pointer is moved each time an address or actual parameter is pushed onto or popped from the stack.
- Notice how the values `Num1` and `Num2` are deleted, not when they are read (popped) but when a new value is pushed.
- Next we must consider what happens if the values are passed by reference. This works in exactly the same way. The addresses of variables are passed so there is no need to return the values via parameters. The procedures or functions access the actual addresses where the variables are stored.
- For functions to return a value, it simply pushes it into the stack immediately before returning. The calling program can then pop the value from the stack. Note that the return address has to be popped off the stack before pushing the return value onto the stack.
- The rule for where is the stack pointer: It points to the **next** item to be read.



8 Recapitulation – Scope of Function in Main Memory during Runtime

- When new module (procedure, function or method) is called (the stack takes the role for the process):
  - the caller is suspended
  - the "state" of caller is saved
  - new space is allocated for variables of the new module
- At the end of new module:
  - release the space allocation.
  - return to the point next to the caller with the previous "state" recovered.
- **With a recursive call, the same occurrences are expected.**

Main memory	
Program segment Fixed size	CS
Global and static variables Fixed size	DS
BSS segment Fixed size	
Free-Heap	ES
Stack segment Activation records with return address and local variables	SS

The stack and heap segments, on the other hand, grow and shrink during program execution.