Implementing function/ procedure calls

(in Pascal and C)

Short contents

- Call code, entry code, return code
- Implementing functions/procedures in Turbo Pascal
  - Memory models
  - Turbo Pascal's memory map
  - Call code – passing parameters
  - Entry code – returning result from functions
  - Return code
- Implementing functions/procedures in Borland C
  - Call code – passing parameters
  - Entry code – returning result from functions
  - Return code

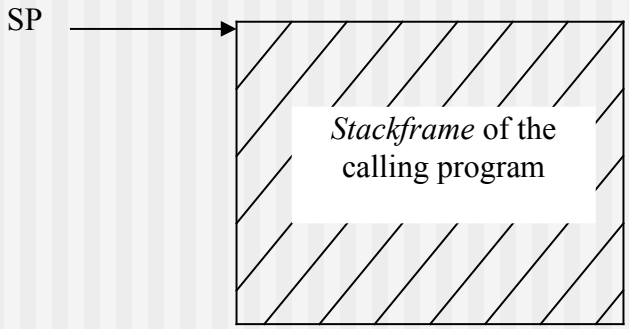
Call code (1)

- Code executed before the actual function call
- Consists of the following actions:
  - If the called code is a function that returns string, the address of the result will be placed onto the stack (segment:offset)
  - Pushing parameters on to the stack
  - Executing a CALL instruction in order to actually call the procedure/function’s code, which pushes the return address onto the stack (far (segment:offset) or near (offset), depending on the type of call)

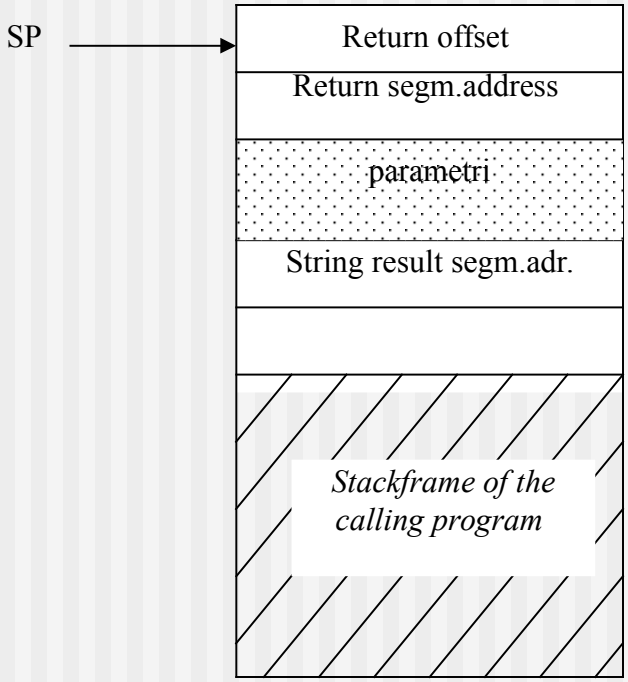
Call code (2)

```
push seg (rez)          ;push the address of the string result onto the stack
push offset (rez)       ;(only for functions returning string)

sub sp, parameters      ;sp's value (the address of  the top of the stack) is updated, substracting
                        ; as many bytes as necessary for holding the parameters onto the
                        ;stack

push seg (return_addr)  ;save the return address on to the stack (far or near)
push offset (return_addr)
jmp function_adr        ;call the function/procedure
```

Call code (3)



The stack before the call



The stack after the call

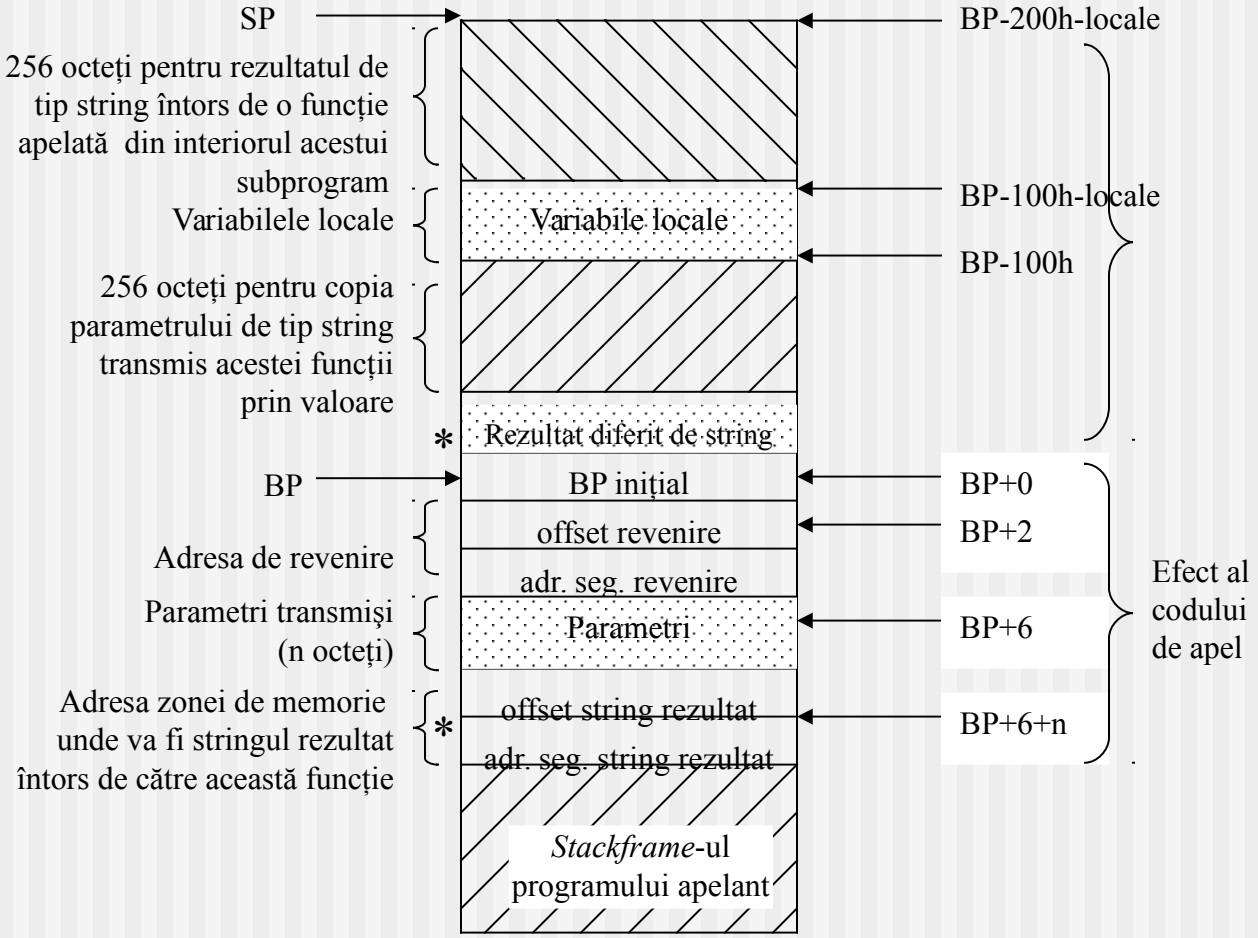
Entry code (1)

- Code executed when entering the function/procedure, prior of executing the first instruction of the function/procedure
- Consists of the following actions:
  - Isolating the stack, i.e. defining a *stackframe* used when executing the called function/procedure's code
  - Reserving stack memory for the result of type different than string returned by this function (if any)
  - Reserving stack memory and locally copying parameters passed by value of size > 4 bytes which were not copied onto the stack by the call code
  - Reserving stack space for local defined data
  - Reserving stack memory for the string type result returned by functions called from this code

Entry code (2)

```
push bp          ; push BP on to the stack, so that it can be restored by the return code
mov bp,sp        ; BP points now to the beginning of the stackframe

sub sp,n         ; reserve stack memory for the result different than string returned by this
                 ; function (n – size of result)

sub sp,100h      ; reserve stack memory for local copies of string parameters passed by
                 ; value
...              ; making local copies into the space reserved previously
sub sp,locals    ; reserve 'locale' bytes onto the stack for local defined data

sub sp,100h      ; reserve stack memory for the string type result returned by functions
                 ; called from this code
```

Entry code (3)



Stack after entering the function/procedure

Return code (1)

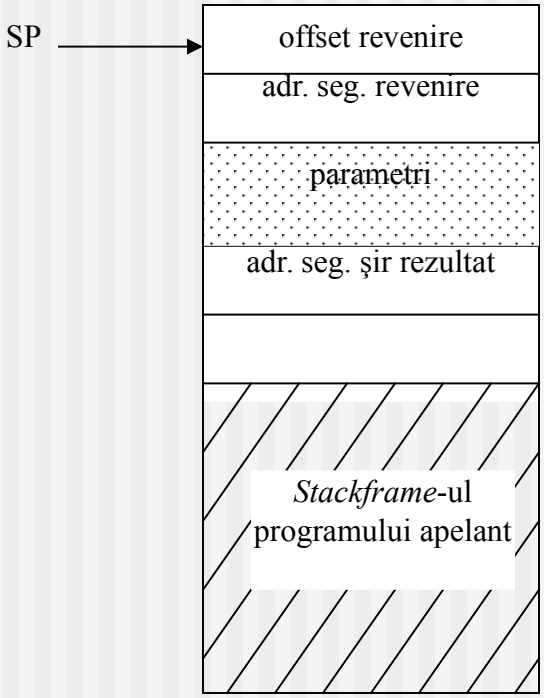
- Code executed when returning from function/ procedure, after executing the function/ procedure's last instruction
- Consists of the following actions:
  - Restoring the stack so that the registers which define it (SS, SP and BP) have the values they had before entering the function/procedure
  - Removing parameters from the stack and returning to the calling code (of course, the return address is also removed from the stack)

Return code (2)

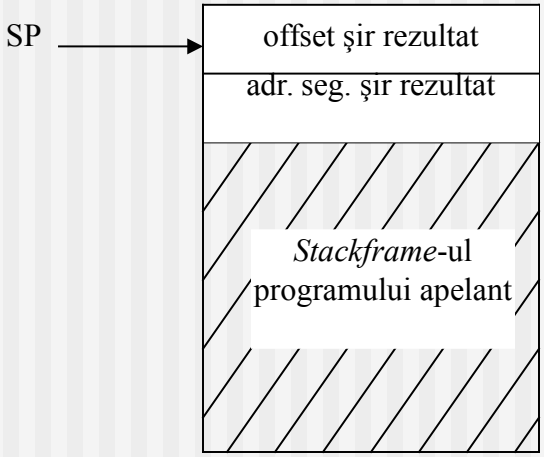
```
mov sp,bp          ; restore SP
pop bp             ; restore BP, free memory space reserved in the stack by entry
                   ; code
ret parameters     ;return from function and remove parameters from the stack
                   ;('parameters' bytes)
```

The calling code has the responsibility of removing the string result's address from the stack (add sp, 004).

Return code (3)



The stack restored

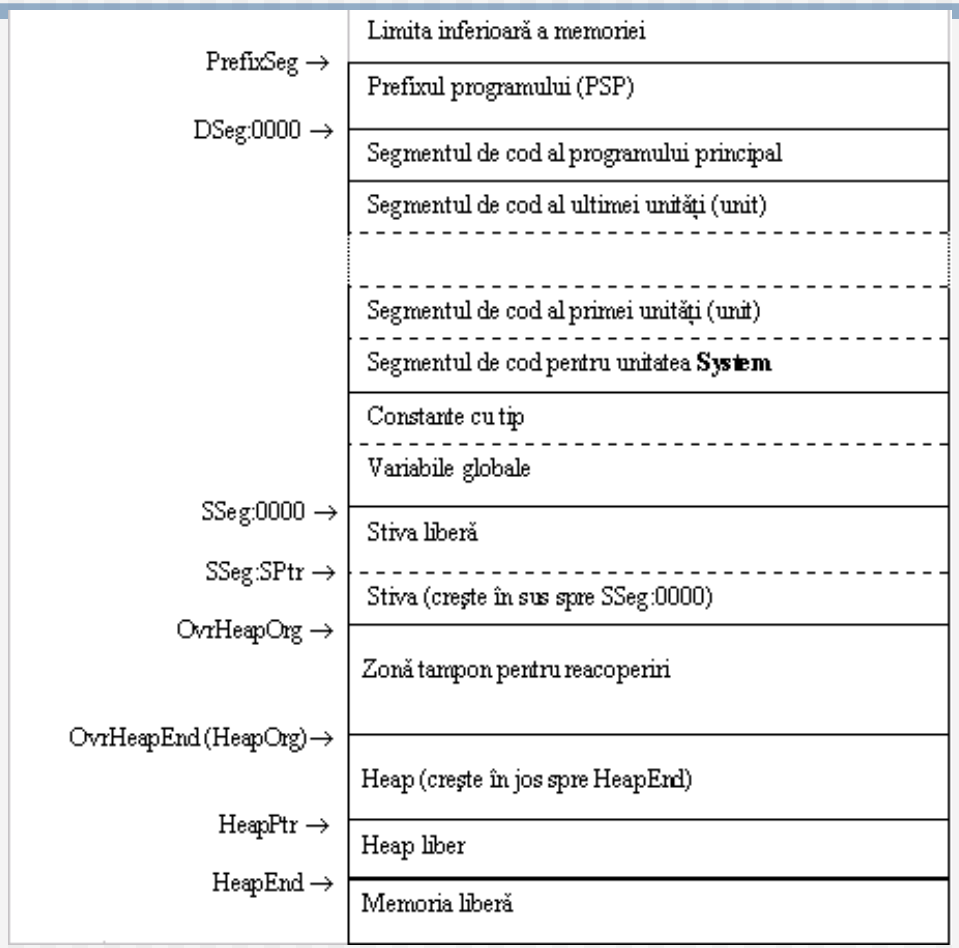


Stack after returning from called code

Implementing functions/procedures in Turbo Pascal. Memory models

- Memory models allowed by Turbo Assembler 2.0:
  - *tiny* : a single segment containing data and code (for .com programs)
  - *small* : a single code segment and a single data segment (most used)
  - *medium* : several code segment, a single segment for data+stack
  - *compact* : a single code segment, several data segments, several stack segments
  - *large* : several data/code/stack segments
  - *huge* : similar to *large* (differ in the size of data that can be defined in the data segment)

Memory map of Turbo Pascal



Upper limit of DOS memory

Call code - passing parameters (1)

- *far* call or *near* call
- parameters are pushed onto the stack in the order they appear, starting from left to right
- for parameters passed by value represented on 1, 2 or 4 bytes the value of the parameter is pushed on to the stack (if the parameter is a byte a word with the value of the parameter as the inferior byte is pushed onto the stack; if the parameter has 4 bytes two words are pushed onto the stack, with the superior word first)
- for parameters passed by reference and for parameters passed by value represented on > 4 bytes (e.g. string, array) their address (far or near) is placed onto the stack

Call code - passing parameters (2)

TYPE	WHAT IS PUSHED ON TO THE STACK
Char	- unsigned byte
Boolean	- byte (value 0 or 1)
enumeration	- unsigned byte, if enumeration has at most 256 values; else, unsigned word
Real	- 6 bytes
floating point value	- 4, 6, 8, 10 bytes on the mathematical coprocessor's stack
Pointer	- 2 words
String	- pointer (far) to value
set	- address of a set on 32 bytes
Array, Record	- value on stack, if the size is 1, 2 or 4 bytes; else, pointer to value

Call code – types of call

- FAR call will be used for:
  - functions defined in the interface section of a unit
  - functions defined after a {$F+} compilation directive or defined using the FAR directive
  - functions defined in the outer most level of a program, if compiling is done with default FAR call option
- NEAR call will be used for:
  - functions defined in the outer most level of a program, if compiling is done without default FAR call option
  - functions defined using the NEAR directive or which don’t come after a {$F+}
  - functions defined in the implementation section of a unit

Entry code – returning result from functions

- integer result:
  - on 1 byte -> AL
  - on 2 bytes -> AX
  - on 4 bytes -> DX:AX
- real result -> DX:BX:AX
- floating point result -> the registers of the mathematical coprocessor
- string result: in a memory area with the address placed (by the call code) onto the stack before the parameters
- pointer result: in DX – segment, in AX - offset

Example

```
    program Example1;
    var AY: Byte;
        AS, S: String;
    Procedure A (X: Integer; var Y:Byte; S: String);
1   Begin              ;entry code in A
2                  Y := Lo(X);
3   end;  ;exit code from A
    Function B(N: Integer): String;
    var T: Real;
4   Begin              ;entry code in B
5                  B[0] := Chr(N);
6   end;   ;exit code from A
7   begin
8                  A(5, AY, AS);    ; A's call code
9                  S := B(7);          ; B's call code
10  end.
```

Call code - example

- Call code generated for line 8 (A(5, AY, AS)):

```
mov ax, 0005h
push ax              ; put X's value on the stack
mov di, 0050h        ; di <- AY's offset into the data segment
push ds              ; put AY's segment address on the stack
push di              ; put AY's offset on the stack
mov di, 0052h        ; di <- AS' offset
push ds              ; put AS's segment address on the stack
push di              ; put AS's offset on the stack
call Example1.A ; the actual call
```

Entry code – example (1)

- Entry code generated when entering procedure A (line 1):

```
push bp
mov bp, sp              ; isolate the stack
mov ax, 0100h
call 689C:02CDh
sub sp, 0100h           ; reserve stack space for local copy of string parameter passed by
; value
mov bx, ss
mov es, bx
mov bx, ds
cld
lea di, [bp-100]
lds si, [bp+4]
lodsb
stosb
xchg cx, ax
xor ch, ch
rep movsb
mov ds, bx              ; actual copy of the string on to the stack
```

Entry code - example (2)

- Entry code generated when entering function B (line 4):

```
push bp
mov bp, sp
mov ax, 0006
call 689Ch:02CDh
sub sp, 0006    ; reserve stack for real type parameter
```

- Call code generated for function B (line 9):

```
lea di, [bp-0100h]
push ss
push di          ; push the address of the string result on the stack
mov ax, 0007
push ax
call Example1.B
add sp, 0004    ; remove the result's address from the stack
```

Return code - example

- Return code from procedure A (line 3):

```
mov sp, bp
pop bp
ret 000Ah       ; removing 10 bytes from the stack
```

- Return code from function B (line 6):

```
mov sp, bp
pop bp
ret 0002        ; removing 2 bytes from the stack
```

Implementing functions/procedures in Borland C. Call code

- FAR or NEAR call depending on the memory model used
- Parameters passed by value only
- Parameters placed on the stack starting from right to left
- 1 word is pushed on the stack for *char, enum, int, near pointer*
- 2 words are pushed on the stack for parameters represented on 4 bytes like *long* and *far pointeri*
- real parameters (*float, double* and *long double*) and *struct* are copied on the stack

Entry code

- Compiler doesn’t have to generate code for local copying of parameters passed by value with a size greater than 4 bytes
- Returning result is done similar to the way Pascal returns result from functions

Return code

- The task of removing parameters from the stack falls to the calling code
- Return code example:

```
mov sp, bp
pop bp
ret
```