ANSWERS

A2.1 Stack frames and the call stack

Recursion and computational thinking · A level · OCR H446 2.2.1, AQA 7517 4.1.1.15 · about 20 min

What this lesson is about

Return addresses, parameters and local variables: what a subroutine call pushes, and what a return pops.

- What has to be remembered
- Pushing and popping frames
- Each call has its own locals
- Modelling the stack yourself
- When the stack runs out

Questions

6 marks in all

1. Which of these does a stack frame store for a subroutine call? [1 mark]

Tick every answer that is true.

- ☐ **A** The return address
- ☐ **B** The parameters
- ☐ **C** The local variables
- ☐ **D** The source code of the subroutine

> **Answer:** A, B, C. A frame holds the return address, the parameters and the local variables for one call. The code itself is stored once, not in every frame.

2. What happens to the call stack when a subroutine returns? [1 mark]

- ☐ **A** The top frame is popped and execution continues at its return address
- ☐ **B** A new frame is pushed
- ☐ **C** The bottom frame is popped
- ☐ **D** Every frame is cleared

> **Answer:** A. The most recent call is the one that returns, so its frame is on top. Its return address says where to carry on.

3. Why is a stack the right structure for subroutine calls? [1 mark]

- ☐ **A** The most recently called subroutine is always the first to finish
- ☐ **B** Subroutines finish in the order they were called
- ☐ **C** It lets any frame be read directly by its position
- ☐ **D** It uses no memory

> **Answer:** A. Calls end in the reverse order to the one they started in, which is last in, first out.

4. What does this program print? [1 mark]

```
def inner():
    total = 100
    return total

def outer():
    total = 5
    inner()
    return total

print(outer())
```

> **Answer:**
>
> ```
> 5
> ```
>
> inner's total is a local in a separate frame, which is popped when inner returns, so outer's total is still 5. The value inner returned was not used.

5. main calls a(), a() calls b(), and b() calls c(). Counting a frame for the main program, how many frames are on the call stack while c() runs? [1 mark]

> **Answer:** 4. One frame each for main, a, b and c.

6. What is the name of the error when a chain of calls uses up all the memory set aside for the call stack? [1 mark]

> **Answer:** stack overflow. Each unfinished call holds a frame, and the stack has a fixed size, so calls nested too deeply overflow it.

The task: watch the call stack

The starter drives the robot through `lap(10)`, which calls `corner` twice, and each `corner` calls `beep`. Make the call stack visible. - Keep a list called `stack`. At the **start** of every subroutine, push a frame onto it: a dictionary with the keys `"name"`, `"params"` (a dictionary of parameter name to value) and `"return_to"` (the name of the caller, or `"main"` for the call from the main program). Then print `push <name> <parameter>=<value>, return to <caller>, depth <d>`, where `<d>` is the number of frames on `stack` after the push. - At the **end** of every subroutine, pop the top frame and print `pop <name>, depth <d>`, where `<d>` is the number of frames left. - Keep track of the deepest the stack gets, and after `lap(10)` has finished print `deepest: <n>`. The first line is `push lap size=10, return to main, depth 1` and the last before `deepest` is `pop lap, depth 0`. The robot still drives and beeps as before.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

stack = []

def beep(freq):
    tone(freq, 0.2)

def corner(cm):
    forward(50, distance=cm)
    beep(880)

def lap(size):
    corner(size)
    corner(size)

lap(10)
```

The hint students can ask for: Every subroutine needs the same two jobs: one at its first line and one just before it ends. Write those two jobs once as helpers. The depth is simply how many frames are on the list at that moment, and the deepest is the largest that ever gets.

A solution

```
from bugbot import *
connect()

stack = []
deepest = 0

def push(name, param, value, return_to):
    global deepest
    stack.append({"name": name, "params": {param: value}, "return_to": return_to})
    deepest = max(deepest, len(stack))
    print(f"push {name} {param}={value}, return to {return_to}, depth {len(stack)}")

def pop():
    frame = stack.pop()
    print(f"pop {frame['name']}, depth {len(stack)}")

def beep(freq):
    push("beep", "freq", freq, "corner")
    tone(freq, 0.2)
```

```
    pop()

def corner(cm):
    push("corner", "cm", cm, "lap")
    forward(50, distance=cm)
    beep(880)
    pop()

def lap(size):
    push("lap", "size", size, "main")
    corner(size)
    corner(size)
    pop()

lap(10)
print("deepest:", deepest)
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.