Rechnerarchitektur Serie 2

Manuel Flückiger 22-112-502

March 27, 2023

Theorie Aufgaben

1 Single-functional for Loop

Is this loop infinite and what will be printed by this code snippet? Explain your reasoning.

```c
#include <stdio.h>

int get_number()
{
    static int number = 8;
    return --number;
}

int main()
{
    for (get_number(); get_number(); get_number())
    {
        printf("%d ", get_number());
    }

    return 0;
}
```

This code will not produce an infinite loop, it will print:

```
5 2
```

Reason for that is the `static` keyword in the function `get_number()` in front of `number`. What that keyword does in that context is, it makes the variable retain it's value among multiple function calls. My reasoning for the output, aswell as the for loop and it's non-infinite runtime:
The initialization statement is given by `get_number()`, which returns 7. This value, however will not be used in the loop itself, it's just important to understand, that it already lowers `number` by 1. The test expression is then also

set to `get_number()`, which means the for loop will run until `get_number()=0`. Once again, `number` will be lowered by 1 everytime the test expression is called. Then, for the update statement we have `get_number()` again, which means everytime the loop is finished, `number` will be lowered by 1. Lastly, we have the code inside the for loop itself, which is just a `printf()` of the return value of `get_number()`, which of course, lowers `number` by 1.
So let's explain the outcome:
before the statements in the loop happen, `get_number()` is called twice, once in the initialization statement, and once in the test expression. `number = 6`
Then the `printf()` happens, in which we call `get_number()` and so we get 5 back. `number = 5`
Then we're at the end of the for loop and the update statement happens, which lowers `number` to 4.
Then the test expresion is tested, from which we get back `number=3`, which is not 0, so we run again.
We get the the `printf()` again, from which we print `number=2`.
For the last time we call the update statement so `number=1`
And now we call the test expresion which now returns a value of 0, which makes it so the expresion is `false` and we exit the loop and finish our programme.

2 Asterisk and Pointifix: Mission Dereference

What will be printed by this code snippet? Explain your reasoning

```c
#include <stdio.h>

int main()
{
    int arr[2][2][2] = {{{1, 2},{3, 4}},
                        {{5, 6},{7, 8}}};
    int x = *(**arr + 1);
    int y = *(*(*arr + 1) + 1);
    int z = **(*(arr + 1) + 1);

    printf("%d %d %d", x, y, z);

    return 0;
}
```

This code will print 2 4 7.
First we need to understand that the asterisk in front of the brackets means dereference. That means, our variable should be set to the value of the address which the then following pointer points to. I'll go trough the pointers line by line.

2.1 `int x = *(**arr + 1);`

`**arr` is a pointer to position `arr[0][0][0]`. To get the value of the position, we have to dereference this pointer, that's what the asterisk in front of the brackets does. We now have to understand which dimension the `+ 1` is added to. It helps, if we use an equivalent display of `**arr` which is `*arr[0]`. Since the index in that display is 0, we know that our element is gonna be in row 0, `arr[0][0][0]`. But we have `**arr + 1` (equivalent to `*arr[0] + 1`) which means we're going to be in column 1, `arr[0][0][1]`. There's nothing changed in the depth of the array index, so our third value is going to be 0 as well (We're still on the first plane of 2D Arrays), so we're looking for `arr[0][0][1]` which in our case is 2.

2.2 `int y = *(*(*arr + 1) + 1);`

Now we're changing the row, since `*(*arr+1)` = `*arr[0 + 1]` and the column is the same as with `x`, `+ 1`. So now we're looking for `arr[0][1][1]`, which is 4

2.3 `int z = **(*(arr + 1) + 1);`

`*(arr+1)` in this case is equivalent to `arr[1][0][0]`, so we've changed the depth we're looking for in the array. Adding `+ 1`, `*(arr+1) + 1` means we're searching at `arr[1][1][0]`, which is 7.

3 Asterisk and Pointifix vs. Incrementor

What will be printed by this code snippet? Explain your reasoning

```c
#include <stdio.h>

int main()
{
    int arr[2][2][2] = {{{1, 2},
                         {3, 4}},
                        {{5, 6},
                         {7, 8}}};

    int(*p)[2][2] = arr;

    int x = *(***+p + 1);
    int y = *(*(*p--) + 1);
    int z = ***p;

    printf("%d %d %d", x, y, z);
```

```
17
18      return 0;
19  }
```

This code will print 6 6 1.

3.1 int(*p)[2][2] = arr;

This is just a declaration of a pointer in a 3D-array.

3.2 int x = *(**++p + 1);

The first asterisk can be ignored once again, as it just tells us to look at the value of the following address. `**p` is like `**arr`, they point at the same memory address. So `****p` is like `**arr[0+1]`, which means, that we're going to be in the second plane of 2D-Arrays. Important to notice, ++ comes before p, so p is incremented before we access it's value. Only thing left is the `+ 1` which means `**arr[1]+1` which leaves us with a pointer to arr[1][0][1], which is 6.

3.3 int y = *(*(*p–) + 1);

The decrement operation is applied only after the pointer is accessed so we know we don't change the plane of 2D-arrays. Like in the last exercise the —+ 1— just means next row. So like `x`, `y` is 6.

3.4 int z = ***p;

The pointer has decremented and is now pointing at the first layer of 2D-arrays again, and with no further addition this means `z = arr[0][0][0]`, which is 1.

4 Asterisk and Pointifix vs. Incrementor

What will be printed by this code snippet? Explain your reasoning

```c
#include <stdio.h>

int main()
{
    char phrase[] = "hello";
    char *p = phrase;

    printf("%s", p + p[0] - p[1]);

    return 0;
}
```

This code will print: lo
Reason for that is that Strings in `c` are Character arrays. The pointer points to the start of the string/char-array. Then in the `printf()` statement 3 is added to the pointer. The reason 3 is added is because of the char values of `p[0] = h = 104` and `phrase[1] = e = 101`. `104-103 = 3`. This means, the pointer now points to `phrase[3]`, from where the rest of the phrase, `lo` is printed.

5 Asterisk and Pointifix vs. Incrementor

What will be printed by this code snippet? Explain your reasoning

```c
#include <stdio.h>

int add(int a, int b)
{
    return a + b;
}

int multiply(int a, int b)
{
    return a * b;
}

int main()
{
    int (*function[])(int, int) = {add, multiply};
    int (*p)(int, int) = *function;

    printf("%d ", (*(p++))(2, 3));
    printf("%d", (*(--p))(2, 3));

    return 0;
}

```

This code will print: 5 5
Reason for that is how pointers and increment/decrement work. We have a pointer `*p` to a function array, which includes the two functions `add()` and `multiply()`. In the first `printf()` statement we call the function on which the pointer currently points at, which is the function at `*function[0] = add()`. So the two integers are added, the output is 5 and `*p` is incremented after it's accessed. So `*p` now points to `multiply()`. In the second `printf()` statement `*p` is decremented before it is accessed, so it will point at `*function[0] = add()` again, and the output will be 5.