Spring 2018

T-501-FMAL, Programming Languages

Third Take-Home Assignment

March 6th, 2018

Edda Steinunn Rúnarsdóttir, kt. 241095-2909

Skúli Arnarsson, kt. 280995-2079

Problem 1. *Consider the following program with exceptions:*

```
1   {
2       int x;
3       read (x);
4       try {
5           x = x + 1;
6           try {
7               if (x == 3)
8               raise E0;
9               else
10              raise E1;
11              x = x * 91;
12          } catch (E0) {
13              x = x + 17;
14          }
15          x = x * 137;
16      } catch (E1) {
17          x = x + 42;
18      }
19      x = x + 1;
20      write (x);
21  }
```

What will be printed if (i) 2 is read, (ii) 1 is read?
Explain.

Solution.

i. When 2 is read, 2741 will be printed. Let's explore the nature of this output by examining this program's behaviour step-by-step using the code as our guide:

- in line 4 immediately after executing the read(x) statement that places 2 as the value of the global x value, we go into the try-block of the outermost try-catch block of the program. When we arrive at this try-block, it's first statement in line 5 changes the value of x:

$$x = x + 1 \ := \ x = 2 + 1 \ := \ x = 3$$

  So now, our x has the value of 3.

- Now in line 6 we arrive at our second try-block which resides within the other try-block, i.e. we enter the try-block of the innermost try-catch block of the program. Now we compare x to 3 (line 7). The x we have indeed has the value of 3 thus this conditional evaluates as true. This results in execution of line 8 which raises the exception *E0*.

- Once *E0* has been raised, we break out of the innermost try-block and the code in lines 9-11 is not executed. We immediately arrive at the catch-block of the innermost try-catch statement. In that catch-block (line 13) the value of our x is changed:

$$x = x + 17 \ := \ x = 3 + 17 \ := \ x = 20$$

  So now, our x has the value of 20.

- now that we've completed the innermost try-catch block we note that we're still in the outermost try-catch block (no exception has been raised in that block that has forced us to break out of it yet). One line remains now in the outermost try-block which changes the value of x (line 17):

  $$x = x * 137 \;:=\; x = 20 * 137 \;:=\; x = 2740$$

  So now, our x has the value of 2740.

- Now that we've ran through the outermost try-block without ever having had to break out of it, we immediately resume our program *after* the try-catch block entire (starting with line 19). Now, the value of x is changed (line 19):

  $$x = x + 1 \;:=\; x = 2740 + 1 \;:=\; x = 2741$$

  So now our x has the value of 2741.

- Immediately after x changes to 2741, the write statement in line 20 is executed on x which explains the output. Therefore, the program outputs **2741** □

ii. When 1 is read, 45 will be printed. Let's explore the nature of this output by examining this program's behaviour step-by-step using the code as our guide:

- in line 4 immediately after executing the read(x) statement that places 2 as the value of the global x value, we go into the try-block of the outermost try-catch block of the program. When we arrive at this try-block, it's first statement in line 5 changes the value of x:

  $$x = x + 1 \;:=\; x = 1 + 1 \;:=\; x = 2$$

  So now, our x has the value of 2.

- Now in line 6 we arrive at our second try-block which resides within the other try-block, i.e. we enter the try-block of the innermost try-catch block of the program. Now we compare x to 3 (line 7). The x variable does not have value 3 thus this conditional evaluates as false so we immediately execute the else statement of line 10 which raises the exception *E1*.

- Once *E1* has been raised, we break out of the outermost try-block (and consequently the innermost try-catch block too, as it's within the outermost try-block) and the code in lines 9-15 is therefore not executed. We immediately arrive at the catch-block of the outermost try-catch statement. In that catch-block (line 17) the value of our x is changed:

  $$x = x + 42 \;:=\; x = 2 + 42 \;:=\; x = 44$$

  So now, our x has the value of 44.

- Now that we've ran through the outermost try-catch block, we move on to the code below it. Now, the value of x is changed (line 19):

  $$x = x + 1 \;:=\; x = 44 + 1 \;:=\; x = 45$$

  So now our x has the value of 45.

- Immediately after x changes to 45, the write statement in line 20 is executed on x which explains the output. Therefore, the program outputs **45** □

Problem 2. *Suppose int is a subtype of float in a language with subtyping and implicit conversions (but no polymorphism).*
If the following program is well-typed, what is the type of the function f?

```
1    int i, j;
2    float x, y;
3    ...
4    i = f (x);
5    y = f (j);
```

Explain.

Solution. Let's put subtyping in context: since an int is defined to be a subtype of a float, it implies that a variable of type int can always be substituted with a variable of type float, but **not** vice-versa (i.e. a float can not necessarily be substituted with an int). This is because float is essentially a supertype of an int (and possibly more types) which further tells us that a variable of type int is essentially also of the type float, but **not** vice-versa (i.e. a float is not necessarily an int). With this intuition in mind, we explore the program to find out the return type of f.

In line 4, the function f takes in a variable of type float as a parameter but it's return value is stored in an int variable. In line 5 however f takes in a variable of type int as a parameter, but it's return value is stored in a float variable. With intuition we can see in this context that f returns an int, but let's explore that further for a more concrete proof.

Towards contradiction we assume that f returns float. In line 5, this is quite straightforward as no implicit conversion occurs - float is returned and the returned value is stored in a float variable. In line 4 however, an implicit conversion must occur between the float return type to an int type in order to store the result in the variable i which is of type int. However, this may not be allowed because not all float variables can be substituted for an int. For example, if the returned value would be a decimal number it could not correctly be stored into an int-type variable (as this program is well-typed we assume that this incorrect conversion is disallowed). We arrive at a contradiction; f cannot return a value of type float.

Now we assume that f returns an int. This way, no implicit conversion occurs in line 4 - an int is returned from f and the returned value is stored in an int variable. An implicit conversion however occurs in line 5, when f returns an int which needs to be stored in the float variable y. This conversion however is OK, since type float is a super-type of type int and an int can always be substituted for a float as explained above. **Therefore, the return type of the function f must be int.**

Additionally by that same logic we note that f cannot take in a variable of type int as a parameter, but must take in a variable of type float as a parameter. This is because when f has the input parameter type of a float and takes in an int, the implicit conversion from an int to a float is OK since an int is substituteable for it's supertype i.e. float. But when f has the input parameter type of an int and takes in a float the implicit conversion is not OK, as a float is not necessarily substitutable for an int, for example if the input parameter is a decimal number it cannot be converted to an int. **Therefore, the type of f's input parameter must be float.**

To sum up, the declaration of the function f would therefore look something like this: □

```
int f(float x);
```

Problem 3. *Given the following function definition in a language with polymorphism and inferred types, find the most general type of the function g in the following program.*

```
g (p, f, x) {
    if (p (x))
        return f (x);
    else
        return x;
}
```

Explain.

Solution. An intuitive explanation of this of the most general type of function g is the following:

- Function g can be denoted as:

  $$< parameters > \rightarrow < return\ value\ of\ g >$$

  The function takes in three parameters; p, f and x which we will now explore in attempt to infer their types.

- In line 2 we note that p is used as a function with the input parameter x. We therefore can infer that p must be able to take in a variable of the same type as x as a parameter. Let's denote the type of x as $\boldsymbol{T}$ for simplification. We further note that p's return value is used to evaluate a conditional if-statement. Therefore, we know that p is a function whose return value is of type $\boldsymbol{Bool}$. To sum up:

  $$p := T \rightarrow Bool$$

- In line 3 we notice that f is called with parameter x from which we can infer that f can take in parameter of type $\boldsymbol{T}$ (type of variable x). We can further infer that f's return value is the same as g's return value (which is still unknown to us at this point). To sum up:

  $$f := T \rightarrow < return\ value\ of\ g >$$

- In line 5 we notice that x can be returned if the if-statement in line 2 is evaluated as false. Therefore function g must return type $\boldsymbol{T}$ (type of x).To sum up:

  $$x := T$$

  which tells us that

  $$g :=< parameters > \rightarrow\ T$$

  And then we can we add to this our definition of g's parameters. To sum up, the formal explaination of g's most general type is denoted by the expression:

  $$g := \forall T (T \rightarrow Bool) \times (T \rightarrow T) \times T \rightarrow\ T$$

□

Problem 4. *Consider a programming language with pointer types and an explicit dereferencing operator. The programmer deallocates memory and there is no safety mechanism implemented to detect the emergence of dangling references. Suppose that assigning to a pointer the value null does not free the memory it pointed to, it merely sets the pointer value to null, ie the pointer now does point anywhere. free frees the memory (without clearing it), but the pointer still points to the old place.*

(i) What does the following program print? Explain.

```
int *p, *q;
p = new int;
*p = 5;
q = p;
p = null;
write (*q);
```

(ii) What will the following program print? Explain.

```
int *p, *q;
p = new int;
*p = 5;
q = p;
free (p);
write (*q);
*q = *q + 1;
write (*p);
```

(iii) What will the following program likely print? (Assume that free heap memory is tracked with a free list, so that memory freed last will be taken into use anew first.) Explain.

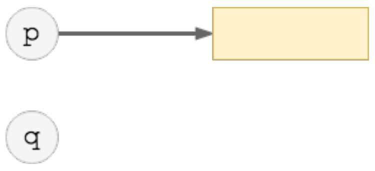
```
int *p, *q, *r;
p = new int;
*p = 5;
q = p;
free (p);
p = null;
write (*q);
r = new int;
*r = 17;
write (*q);
```

Solution.

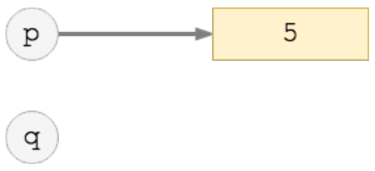
(i.) The program outputs: 5. Let's explore this program's behavior via pointer diagrams for better understanding.

- In line 1 of the program segment, pointers p and q are initialized:

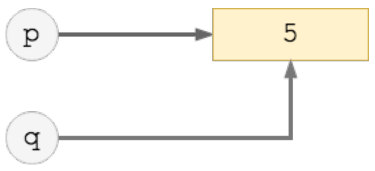
- In line 2 of the program segment, p is pointed to a new integer:



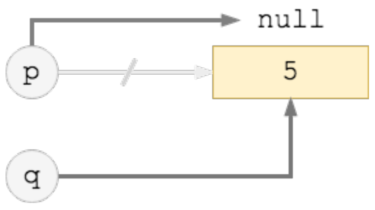
- In line 3 of the program segment, the value p points to is set to 5:



- In line 4 of the program segment, the pointer q is pointed to what p points to:



- In line 5 of the program segment, p is pointed to null:



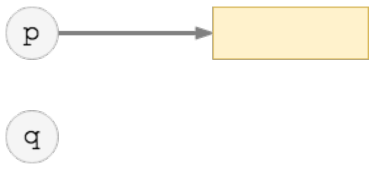
- In line 6 of the program segment, q is written out and as illustrated in diagrams above, q has the **value of 5**.

Therefore this program segment ***outputs 5***.

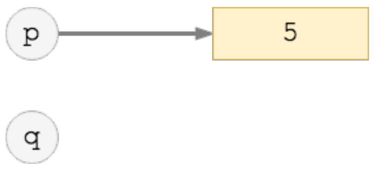
(ii.) The program outputs: 5 6. Let's explore this program's behavior via pointer diagrams for better understanding.

- In line 1 of the program segment, pointers p and q are initialized:

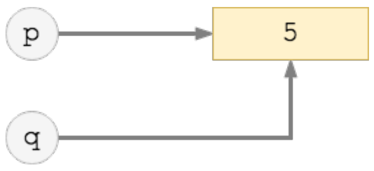
- In line 2 of the program segment, p is pointed to a new integer:



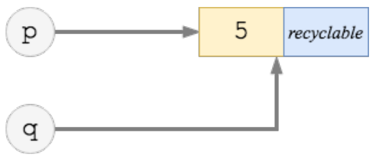
- In line 3 of the program segment, the value p points to is set to 5:



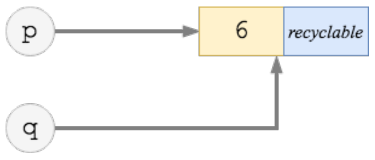
- In line 4 of the program segment, the pointer q is pointed to what p points to:



- In line 5 of the program segment, p is freed. But as specified in problem, when a pointer is freed it still points to the old place, and the old memory slot is marked to be 'recyclable' i.e. informing that this specified memory slot can recycled if needed:



- In line 6 of the program segment, q is written out. Since no allocations of values have been conducted since the free(p) statement, the memory containing the value that q points to has not yet been recycled despite having been marked recyclable by the program. Therefore, q has the **value of 5** so 5 is outputted.
- In line 7 of the program segment the value that q points to is incremented by one, so the memory slot to which q points becomes 5+1 = 6:



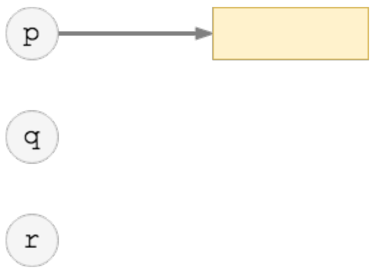
- In line 8 of the program p is printed out. Now we note that again no allocations of values have been conducted since the free(p) statement, so the memory containing the value that p points to (p still points to the memory slot it did point to despite having been freed) has not yet been recycled despite having been marked recyclable by the program.

Therefore this program segment ***outputs 5 6***.

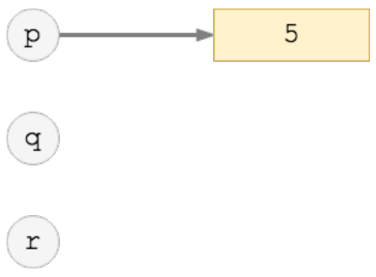
(iii.) The program outputs 5 17. Let's explore this program's behavior via pointer diagrams for better understanding.

- In line 1 of the program segment, pointers p, q and r are initialized:

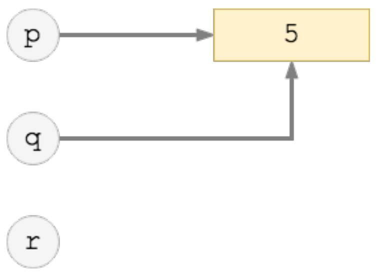
- In line 2 of the program segment, p is pointed to a new integer:



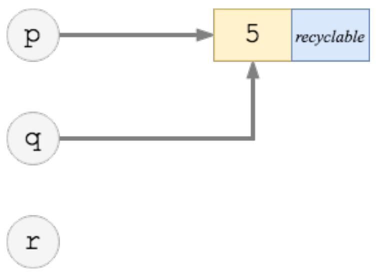
- In line 3 of the program segment, the value p points to is set to 5:



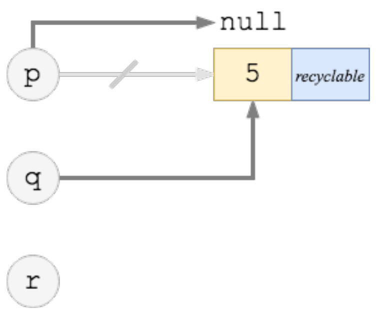
- In line 4 of the program segment, the pointer q is pointed to what p points to:



- In line 5 of the program segment, p is freed. But as specified in problem, when a pointer is freed it still points to the old place, and the old memory slot is marked to be 'recyclable' i.e. informing that this specified memory slot can recycled if needed:

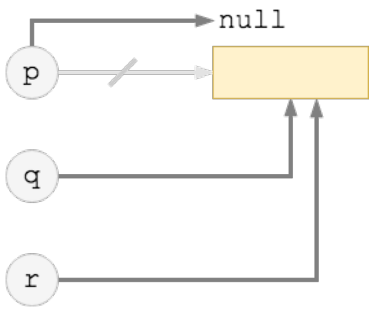


- In line 6 of the program segment, p is set to null:

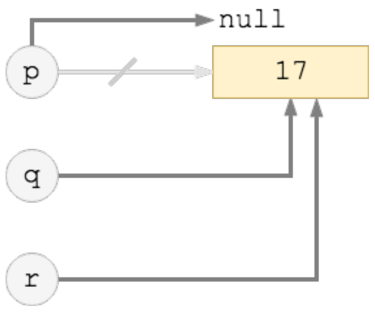


- In line 7 of the program segment, q is written out. Since no allocations of values have been conducted since the free(p) statement, the memory containing the value that q points to has not yet been recycled despite having been marked recyclable by the program. Therefore, q has the **value of 5** so 5 is outputted.

- In line 8 of the program segment, r is set to a new int. Now since we assume that free heap memory is tracked with a free list, this new int uses the recyclable memory address that p previously pointed to and freed. However incidentally since pointer q still points to p's previous memory location, this assignment essentially changes q's assignment:



- In line 9 of the program segment, r's is set to 17 which as described not only changes the value r points to but the value q points to, as r and q point to the same memory location:



- In line 10 q is printed out. q now has the value of 17 as shown in diagram above so the program outputs 17.

Therefore this program segment ***outputs 5 17***.

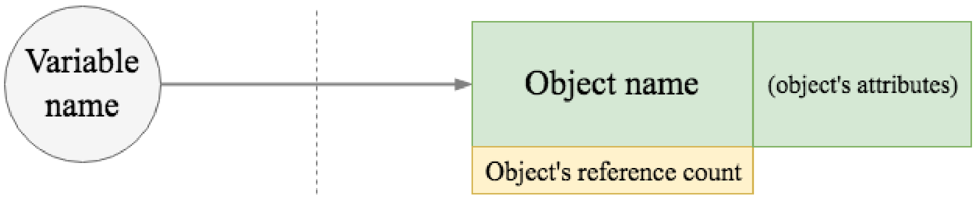
Problem 5. *Suppose we have a language using the reference model (i.e. all variables are references, there is no explicit dereference operator). Memory is freed by a reference-counting garbage collector.*
Consider the following program. State which variables point to which objects after the execution of each line of the program and what the values of the reference counters of each object are (you can show only the information that changed during the execution of the line).

Which objects can be garbage-collected after the execution of the last line?

```
type C = ...;
type D = record {C f; ...};

C c1 = new C;        //o1 is allocated (and c1 initialized to point to o1)
C c2;                //c2 is initialized to null
D d1 = new D;        //o2 is allocated (and d1 initialized to point to o2)
D d2 = new D;        //o3 is allocated
d1.f = c1;
c2 = c1;
c1 = new C;          //o4 is allocated
d1.f = c1;
d2.f = c2;
c2 = null;
d2 = d1;
```

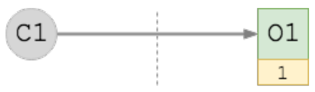
Solution. The following diagrams show how the state of the variables and objects line-by-line. The diagrams all have the following format that clarify the states and of each variables, object and references:



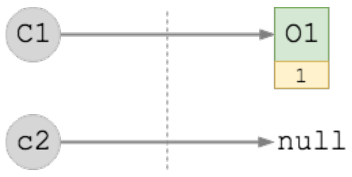
That is, the variables are placed on the left side of the figure and point to objects on the right side. Object can have zero, one or more attributes that are placed on the object figure's right, and each object has a reference count denoting how many references it has. Objects can of course point to other objects or object attributes, and object's attributes can point to other objects or object attributes.

The following demonstrates the states of variables, objects and reference counts at each line of code:

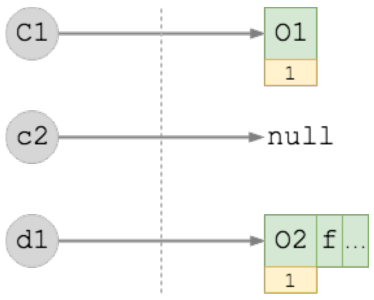
- line 4 int the code creates a new C object, c1, that points to o1:



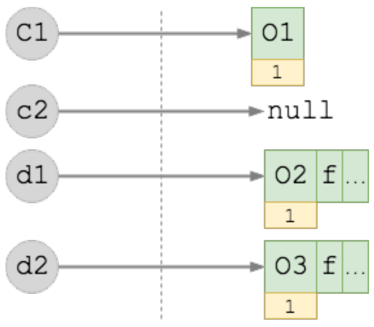
- line 5 in the code creates a new C object, c2, that is initialized as null:



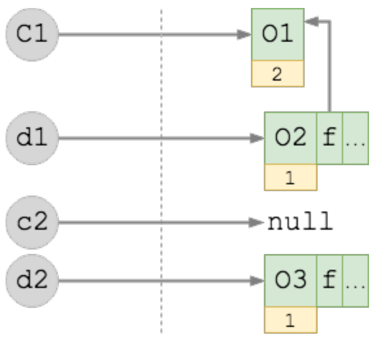
- line 6 in the code creates a new D object, d1, that points to object o2 which of course has it's own f-attribute (which is a c-object) and more attributes since it's a D-object:



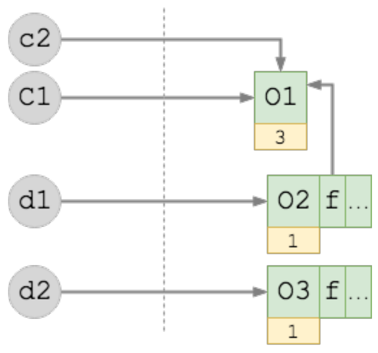
- line 7 in the code creates a new D object, d2, that points to object o3 with it's own f-attribute:



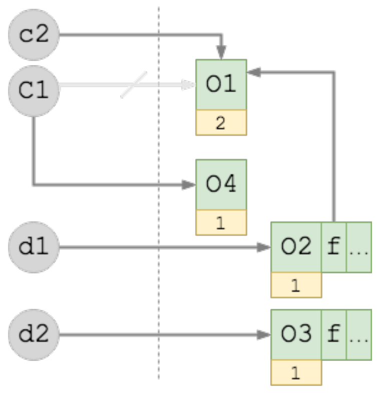
- line 8 in the code sets the f attribute of d1 to c1. d1's f attribute therefore points to c1:



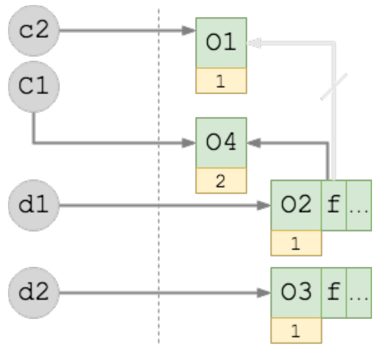
- line 9 in the code has the c2 object point to c1:



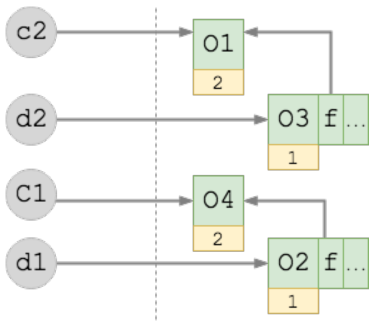
- line 10 in the code sets c1 to a new object, o4:



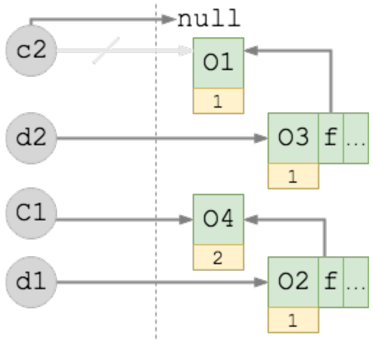
- line 11 in the code sets the f-attribute of d1 to c1:



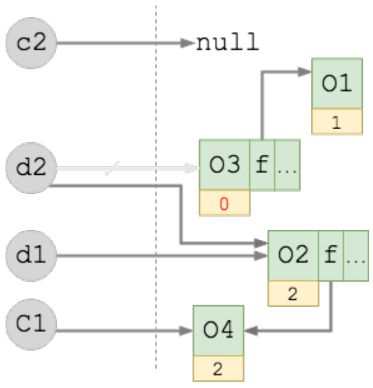
- line 12 of the code sets the f-attribute of d2 to c2:



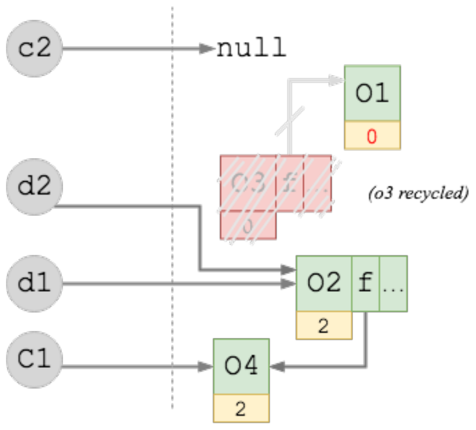
- line 13 of the code sets c2 to null:



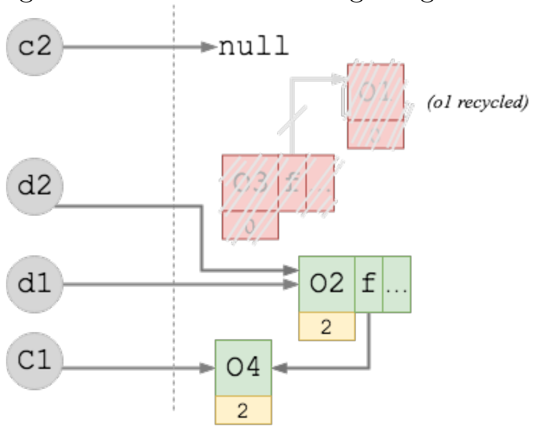
- line 14 of the code lets d2 point to what d1 points to:



Now that the code has run we notice that since reference count to object o3 has gone down to zero it can be garbage collected:



Furthermore as displayed in above figure, after garbage-collecting o3, o1's reference count has also gone to zero so o1 can be garbage collected as well:



Therefore, ***the objects o3 and o1 can be recycled when the code has run***.