Concepts of Programming Languages (INFOTECH)

Summer Semester 2006
Steffen Keul, Stefan Staiger

Solutions to Assignment #2

1 Parameter Passing Mechanisms

1.1 Value of a after the call $p(a, a)$

Call by value x and y are independent from a, so modifications to them have no effect on the contents of a. When p is called, the contents of a are copied into x and y, but there is no copy-back that returns anything. So a is left unchanged, $a = (1, 2, 3, 4, 5)$.

Call by reference Now x and y are references to a. Therefore, modifications to them also modify a. We say that a, x, and y are aliases. The loop now actually performs the following steps:

a. $(i = 1)$ $x(1) := y(5)$ now also modifies $a(1)$, so we now have $a = (5, 2, 3, 4, 5)$
b. $(i = 2)$ now also modifies $a(2)$, so we have $a = (5, 4, 3, 4, 5)$
c. $(i = 3)$ basically has no effects
d. $(i = 4)$ now actually reads $a(2)$ and modifies $a(4)$, so the contents a are indeed left unchanged!
e. $(i = 5)$ is similar, so that we finally have as result: $a = (5, 4, 3, 4, 5)$

Call by value/result Here we have the following behaviour:

a. Contents of a are copied into new variables x, y
b. Procedure p results in $x = (1, 2, 3, 4, 5)$ and $y = (5, 4, 3, 2, 1)$
c. Now copy-back is performed, but both x and y must be copied back to a. Therefore, the final contents of a dependent on the order: If x is first copied back, then a will contain the contents of y, that is $a = (5, 4, 3, 2, 1)$. But if y comes first, we will have $a = (1, 2, 3, 4, 5)$. So if the language doesn't guarantee the order here, we can't tell what the final contents of a will be!

Call by name In this case the names x and y are textually replaced with a, and then the name a is bound (which in this case binds to the global variable a). Therefore, the body of the loop is now

$$a(i) := a(6 - i)$$

and the program in this example behaves as seen in the call-by-reference case above.

1.2 Aliasing

Aliasing means that different names (variables) refer to the same location. Thus, if we have aliasing, an operation on one of these names also changes the value of the other "aliases" for the storage location! As we saw, call-by-reference and call-by-name caused aliasing in the example program. Here is another example, this time in C:

```
void main ()
{
    int n, *p;
    p = &n;
    *p = 4;
    printf ("%d", n);
}
```

In this example, `*p` and `n` (or `p` and `&n`) are aliases. The program will print 4.

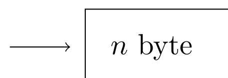
1.3 Mechanisms for the parameters

- y is an "in" parameter, a good choice for it would be call by value.
- x is an "out" parameter and we can use call by reference, call by value/result or in this example even call by name. Probably call by reference would be preferred.
- Alternatively, we could pass x by value/result and y by reference or by name.

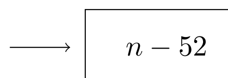
2 Heap management

Type T_1 occupies 20 byte per instance, and type T_2 occupies 32 byte per instance. Let's see what happens with the freelist step by step:

- Before the loop executes, we have one large block of free memory, so the freelist looks like this:

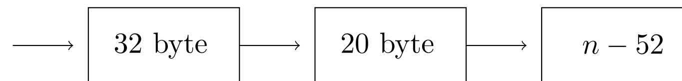


- In the first execution of the loop body, after the allocations for `P` and `Q` we have:

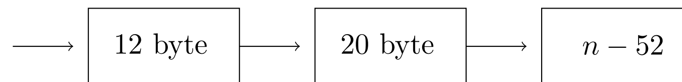


- In the first execution of the loop body, after `Free(P)` we have a new free block:

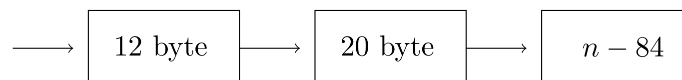
- In the first execution of the loop body, after `Free(Q)` we have a another new free block:



- Now the second execution of the loop body searches for a block of at least 20 byte. Thus, it takes the free storage for the new allocation to `P` from the first block:



- Next, we have to allocate 32 byte for the new instance of T_2. Only the last block is (hopefully) large enough for this request! So we get:



- Freeing P and Q prepends two blocks (of size 20 and 32 byte, respectively) to the freelist, as in the first execution of the loop body.

- Further executions of the loop body repeat this behaviour. Every such execution adds to blocks to the freelist. Finally, the blocks in the freelist have the following sizes:

$$32, 20, 12, 20, \dots, 12, 20, \text{ (all the rest).}$$

  The freelist will have $1 + 100,000 * 2 = 200,001$ blocks and the small blocks will contain

$$32 + 20 + 99,999 * (12 + 20) = 3,200,020 \text{ byte.}$$

As we can see, the allocation `Q := new T2` always searches the (ever growing) complete freelist until the last block is found. This results in a slow down of the program, and perhaps the program even runs out of memory and stops (namely, if n, the size of the large block, is too small).
It is easy to correct the problem: Just exchange the order of the Free statements to

```
Free (Q);
Free (P);
```

and everything will work fine. Alternatively, we could use another strategy to manage the freelist, for example best-fist or the bitvector scheme. And of course we could use an implementation that merges adjacent free blocks.
The bitvector scheme automatically merges adjacent blocks and thus it doesn't have this problem.