Universitetet i Oslo
Institutt for Informatikk

Reliable Systems
Martin Steffen, Gianluca Turin

INF 5110: Compiler construction

Spring 2021

Series 7

12. 3. 2021

Topic: Run-time environments (Chapter 7) (Exercises with hints for solution)

Issued: 12. 3. 2021

Exercise 1 (Run-time environment) Draw a possible organization for the runtime environment of the following C program, for the following two situations. See corresponding figures from the lecture as inspiration (for example, the slide entitled “Stack gcd”, approximately at slide 8.18):

1. after entry into block `A` in function `f`.
2. after entry into block `B` in function `g`.

```
int a[10];
char * s = "hello";

int f(int i, int b[])
{ int j=i;
 A:{ int i=j;
     char c = b[i];
     //...;
   }
   return 0;
}

void g(char * s)
{ char c = s[0];
 B:{ int a[5];
     // ...;
   }
}

main ()
{ int x=1;
   x = f(x,a);
   g(s);
   return 0;
}
```

Solution: We are concentrating on the run-time environment of a C-like language, so we concentrate on the *stack* of activation records (or stack frames) plus space for the *global data.* Not included in the picture is the static part which contains the "code" as it is typically not considered as part of the run-time management, as least not in languages with a very clear separation of "data" and "code" as here in C. Also not considered here is the *heap* even if the example includes a heap-allocated piece of data in the form of the array `b[]` (it's just not part of the exercise).

To solve the exercise, we need to

- figure out, depending on the "kind" of user data, where it's allocated (globally vs. on a stack frame). By "kind" I mean both the type of the data as well as it's role (such as formal parameters of a procedure or locally defined variable of a particular type). The example has data of various different types and in different roles. Furthermore,
- we need a grip on the general *layout* of an AR, i.e., the additional information needed to "make the RTE work". In a C-like language, that includes the following:
  - return address, the
  - control link, as well as the
  - frame pointer, and the
  - stack pointer.

It's important to realize that for a *C-like language,* i.e. without nested procedures and related complications, no *access links* (aka *static links* are needed. They should *not* show up in the ARs. It's important to connect the frame pointer "`fp`" the control link: the frame pointer is the "anchor" or "fix point" of a stack frame, i.e., the point of reference from which the other slots have a known offset. In other words: the frame pointer *represents* the frame, and the control link, sometimes called the dynamic link, points to the "caller's AR" which is represented by the corresponding "anchor" in turn. That means, the control link of the current frame (pointed at by the frame pointer) points at the previous control link etc., i.e., the control links form a *chain.* In the concrete example here, the frame of the main function does *not* indicate any control link, so we simply let the control link of the AR of `f` to point some place "inside" the frame of for the main function. See the "purple section" in Figure 1.[1]

In the graphical representation of the layout, pointers to a slot points to the "bottom" of it. In the picture thus, it's intended to point at the control link.

The other important standard slot in a stack frame is, obviously, the *return address.* It's needed so that the execution (with its instruction pointer) find its "way home from the callee to the caller" upon return. To do that, the caller's value of the instruction pointer has to be saved somewhere, and due to the LIFO nature of function calls, it's saved on a/"the" stack. The so-called *calling convention* of an architecture tells how and where to place that return address in the stack frame. At any rate, it should be kept "close" to the frame pointer (or at least at a fixed off-set), and in the conventions in the lecture, it's the slot following the frame pointer/control link.

Another point in the memory layout of the RTE concerns the *string literal* `"hello"` and the corresponding variable `s` in the example.[2] The variable `s` is declared globally, so the string

[1] It is to be expected, that also the "purple" stack frame for the main function has a "standard" stack frame layout. That would include a slot for a return address and a slot for a control link. It should be arranged in such a way, that the return address is not just "nil", but a designated address that deals with program termination in a dignified way (more dignified at least than trying to execute a nil-pointer). Those details are not part of the discussion in Louden or of this exercise.

[2] The quotation marks technically don't belong to the string literal, but using those marks is the conventional way to demarcate such literals in texts like here in this exercise.

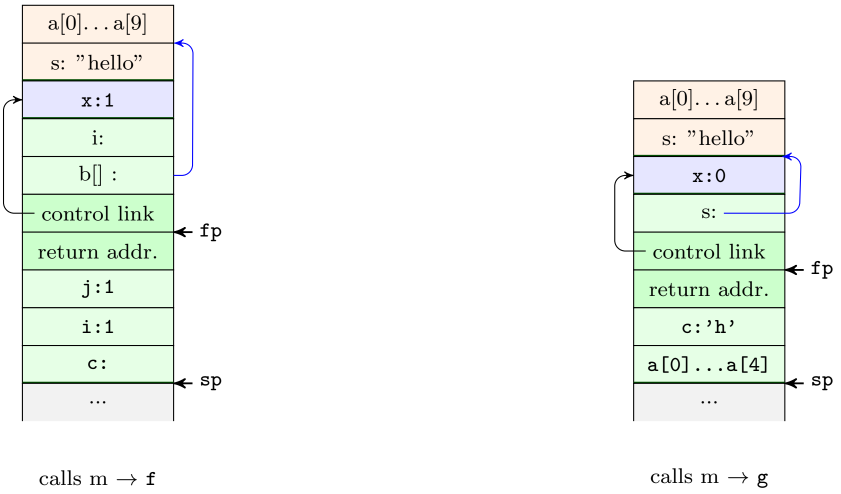


Figure 1: RTE after entering block A (left) resp B (right)

literal ends up there.[3] The same holds for the slots of the array `a`.

See also the "blue arrows". They don't represent "standard slots" in the activation records, but the use of "reference data".

□

Exercise 2 (Activation records (Pascal)) Draw the stack of activation reords for the following *Pascal* program, showing the *control* and *access* links, after the second call to procedure `c`. Describe how the variable `x` is accessed from within `c`.

```
program env;

procedure a;
var x: integer;

    procedure b;
        procedure c;
        begin
            x := 2;
            b;
        end;
    begin (* b *)
        c;
    end;

begin (* a *)
    b;
end;

begin (* main *)
    a;
end.
```

Solution: A crucial thing here is that we are dealing with "Pascal", not "C", respectively the fact that we have nested procedures, which are procedures defined inside others. That's different

[3] As a side remark: string literals are often treated special. Even in situations where a variable containing them were *not* defined globally as in the example, the *string literal itself* would often end up in some global part of the memory where the local variable contains a pointer to it. String literals can be of arbitrary size (but of statically known size) and thus are often considered best placed outside the stack frame.

from the set-up of the previous exercise, where no nesting of procedures was allowed (which reflects the situation in C). Why is that a problem? We are dealing typically with languages with static, lexical scoping (for variable declarations). In general, a procedure or function is well allowed to use variables declared outside the procedure, which means, they are declared in the scope(s) surrounding the function definition. Those are variables in a procedure body neither corresponding to the procedure's formal parameters nor the local variable declarations. Hence they are sometimes called *non-local* variables (more precisely: "procedure-non-local", to distinguish them from the procedure-local ones; there are also distinctions between process-local vs. process-non-local declarations etc.).

In a C-like setting, without nested procedures, the *only* non-local variables are the global ones. Now, however, besides local variables (whose values are are kept in the corresponding AR/stack frame) and global ones, there are *non-local* variables which are not global, as explained.

If a function body uses such a non-local (but non-global either) variable, the question is: where to get its value from? The static *binding* is clear (according to the rules of lexical scoping): the variable which is "meant" has been declared at the closest enclosing scope. Being non-global, as we assume, it has been declared as local to some procedure inside which the procedure we are currently considering has been defined (directly or indirectly). That, however, only clarifies the "static situation" of lexical scoping. When considering the run-time environment, the question is where, at run-time, to *find* the value of a non-local variable, i.e. how to *access its content.* Statically, we know in which surrounding function the variable has been declared, but that it the static aspect only. What is needed is the *activation record,* i.e., the incarnation of that surrounding function, which gives the value to the non-local function. In a language with nested functions (but without higher-order functions), we know that the data of a function call (= the call's activation record) *cannot outlive the activation of the function itself.* That allows the stack-allocation of activation records, which are therefore also known as *stack* frames.

A non-local variable in a stack frame of a function with a has two aspects:

1. static, in which surrounding function is the non-local variable declared and
2. which is the "relevant" activation record, where to find the current value.

Due to the stack-discipline of the AR allocation, the answer to the last question is: the activation record is still found on the stack,

> and *it's the "most recent" stack frame of the surrounding function; a pointer to that stack frame is the access link (or static link).*

That's the general picture. Now to the concrete setting. The first thing to do is to get an overview of the call graph. The program is not really meaningful, the functions `b` and `c` call each other without "exit", which will lead to a stack overflow. But the task asks about the *second call to* `c`. So, the call-sequence is as follows:

$$\mathtt{a} \rightarrow \mathbf{b} \rightarrow \mathtt{c} \rightarrow \mathtt{b} \rightarrow \mathtt{c}$$

The variable in question is `x`, which is declared in `a`. Informally, it's rather simple. We are inside the stack-frame for `c`. It's clear that what is asked for is the value of `x` as being set inside `a`. However, the exercise asks to delineate the access links that make that happen.

So, the access link for the stack frame of the second call to c points to last stack frame of the statically surrounding function, which is the latest stack frame of `b`.

□

Exercise 3 (Access chaining vs. display) An alternative to access chaining in a language with local procedures is to keep the access links in an array *outside* the stack, *indexed* by the *nesting level.* This array is called the *display.* For example, the run-time stacks of the program

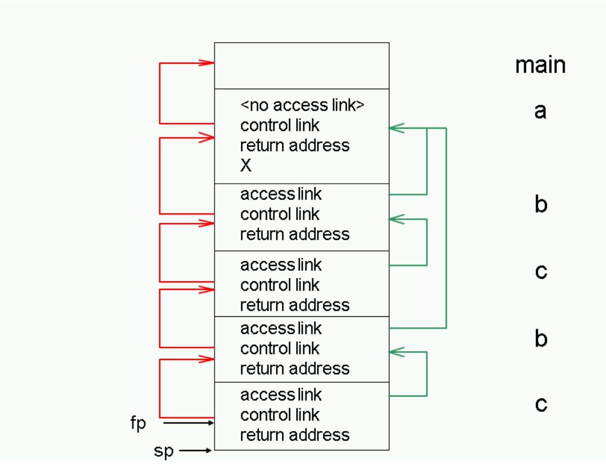


Figure 2: Pascal RTE: control and access links (dynamic and static links)

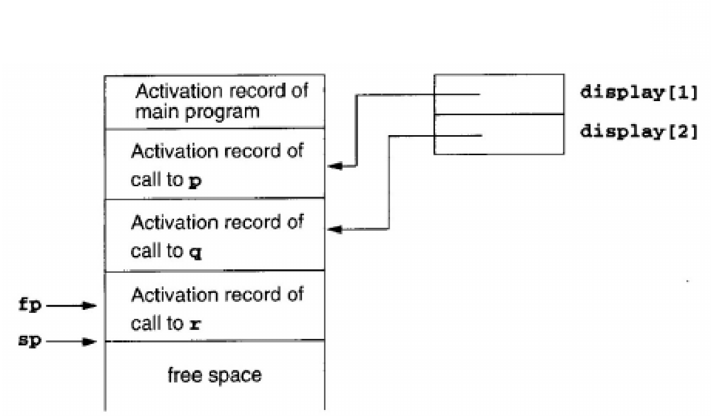

Activation record of main program
Activation record of call to p
Activation record of call to q
Activation record of call to r
free space
fp
sp
display[1]
display[2]

Figure 3: RTE with display (1)

`chain` and the corresponding stack picture on the slide entitled "access chaining" at approx. 8-36 fromx the lecture would now look as Figure 3 resp. Figure 4.

1. Describe how a display can improve *efficiency* of nonlocal references from deeply nested procedures.

2. Redo Exercise 2 from this sheet, using a *display*.

Solution: For access links, there is the phenomenon of *access link chaining*. The difference in static nesting levels from where a variable is declared to where it is use translates to the *number* of "hops" one has to follow at run-time to find the appropriate stack frame where to look up the current *value* (not the declaration) of the variable in question. Note that the *number* of link-followings is statically determinable, but the links themselves are not known (they point to frames of the *run-time* stack ...) and have to be followed at run-time. For each access of such nested variables, there will be a number of indirections to be performed. It's clear that if a significant amount of variable accesses relates to non-local ones with deep link chains, that may result in a serious performance penalty. A display then allows a "two-stop" access: look-up the display in an *indexed* way (which is fast) and follow that address. That may be faster, but of course the display needs to be maintained at run-time. But then again: also the static links need to be maintained within the stack frames, which also does not come for free.

Now, the display is here described as *indexed* by the *nesting level.*

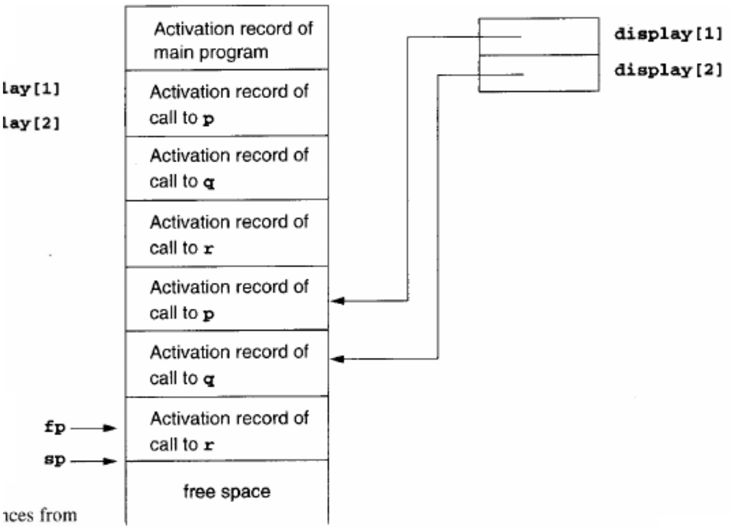
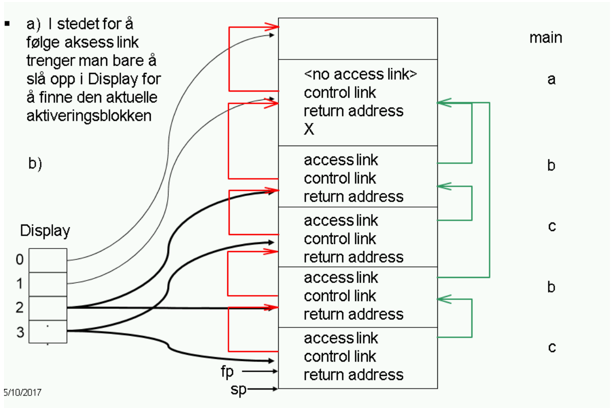

Activation record of main program
Activation record of call to p
Activation record of call to q
Activation record of call to r
Activation record of call to p
Activation record of call to q
Activation record of call to r
free space
lay[1]
lay[2]
fp
sp
ıces from
display[1]
display[2]

Figure 4: RTE with display (2)

A solution is given in Figure 5. The picture may seem a bit confusing.[4] Actually it contains *two* states of the display wrapped into one. This "overlay" is done for the **boldface** arrows (starting from the slots 2 and 3 of the display, for the activation records for activations for `b` and `c`). The *green* arrows are the access links (taken from a previous picture) and are therefore not needed anymore. They are just still shown for better comparison.


- a) I stedet for å følge aksess link trenger man bare å slå opp i Display for å finne den aktuelle aktiveringsblokken

b)

Figure 5: RTE with display

□

Exercise 4 (Virtual function tables and memory layout for classes) Draw the memory layout of objects of the following C^{++} classes, together with the *virtual function tables*.

```
class A
{ public:
  int a;
  virtual void f();
  virtual void g();
};

class B : public A
{ public:
```

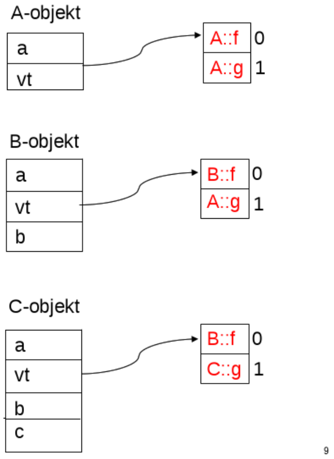
[4] For an exam, it's not a good picture, better make two.

```
  int b;
  virtual void f();
  void h();
};


class C: public B
{ public:
  int c;
  virtual void g();
}
```

Solution: The principles of VFT for standard OO languages (with late binding and class inheritance) have been discussed in the lecture. One needs to know what late binding is and what “standard” methods are. In Java that’s the non-static ones, here they are marked as *virtual*. In practical settings and real programs, the water can be further muddied by *method overloading*. We do not consider that, especially not in this exercise.

The concept of a VTA is pretty simple. It’s just a (often) smart way of helping the run-time system to find the method bodys of methods of an instance of the class. The table keeps an overview of that information *per class*. Each object contains a point to the class it is an instance of (`vt` in the picture). The class corresponds also the the *run-time* type of the object.[5]



□

Exercise 5 (Parameter passing) Give the output of the following program (written in C syntax) using the 4 parameter passing methods discussed in in the lecture.

```
#include <stdio.h>
int i = 0;

void p(int x, int y)
{ x += 1;
  i += 1;
  y += 1;
}

main ()
{ int a[2] = {1,1};
```

[5] Note that in C^{++}-like languages, class names play the role of types. One can distinguish run-time types from static types. It’s not either-or, in Java, C^{++}, one can distinguish, especially for class types, the notion of static type from dynamic (run-time) types. Here, we are speaking about run-time types.

```
  p(a[i],a[i]);
  printf(”%d␣%d\n”,a[0], a[1]);
  return 0;
}
```

Solution: The example procedure has 2 formal parameters, x and y, and makes use of a non-local variable i. Unlike in the exercises concerning the static links, the non-local variable is not defined in a surrounding *procedure* (which is not possible in C anyway), but globally. The variables are not only *accessed* in the function body, but *changed.*

The function has no return value, the result is "passed back" via the side effect of changing the array. Another ingredient in the example is that the actual parameter are array *expressions* and that there are two arguments (not just one). The array expression arguments will play a role for the call-by-name case, the fact that we have two arguments is problematic specifically for the call-by-value-results case.

As a side remarks: arrays are "by-reference" data structures. That fact does *not* play a role in this exercise here, as the arguments to the function is not the array, but expressions of the form `a[i]`, which represent *integers*

Call-by-value: That's the easiest case. The values of the two actual parameters are *copied* into the procedure body. Concretely, upon call, a new stack frame is allocated ("push") with space for the two formal parameters `x` and `y` wherein the *values* of the actual parameters are copied, i.e., a pair of 1's. Whatever the function body does with those, it's done on a copy. Therefore, the original arguments are *unaffected.* That gives the result $(1, 1)$.

Call-by-reference: Now, no copy is done in the new frame, but *references* to the arguments. What is specific in this example is: the two arguments are identical (here `a[i]`). They are not evaluated in the call, instead the *address* of `a[i]` is taken which happens to be the same for both arguments. That it's the same address is independent also from the fact where the array is allocated. We have seen that arrays may be stack allocated or else on the heap, or also sometimes in the static area when they are global and their size is statically known. Independent from all of that, both arguments *have the same address.*

It should also be noted (as a side remark, on a *variation* of the shown code): in a call of the form `p(a[i],a[j])`, both arguments may represent the same address, namely if `i` and `j` happen to be the same. That `i` = j may of course not be clear at compile time, but the point is: the discussion here does not depend on the fact that the two arguments are *literally identical.* That will play a role for "call-by-name" scheme later, but here it's about addresses. The situation, where two different "names for data" (like `p[i]` and `p[j]`, but also in case of variables containing references etc.) refer to data at the same address is called *aliasing*: `p[i]` and `p[j]` are aliases if i and j have the same value. In languages with reference data (objects, etc.), aliasing is not statically determinable in general (of course in this example here, it's obvious).

Aliasing for *immutable* data (as in functional/declarative languages) is not "problematic", problems sometimes starts if there's aliasing in combination of side-effects. This is exactly what is done in the example. Note: the programmer of the procedure uses two different formal parameters (one is not allowed to use 2 times x anyhow) and may conceptually have intended to independently update them "both" by one and might not have forseen that the function is *used* in a way that leads to aliasing (in a call-by-reference setting). If that were the programmer's intention, then the unforseen, accidential aliasing by the user (= caller of the function) "broke" the code.

Anyhow, the corresponding slot in the array is incremented thus 2 times, resulting in the pair $(3, 1)$. Note further, that the variable i is irrelevant when evaluating the body of the function.

Call-by-value-result This parameter-passing scheme one has fallen from grace. Besides the reasons we said in the lecture, the example here is *especially* confusing.

The general principle of call-by-value-result is

1. paramater-passing when *calling* is done like call-by-value (nothing's wrong with that part), and
2. passing the result back when returning: "by reference" (it's this part which is troublesome).

So instead of the more dignified use of a function in the form $x_1 := f(x_2)$, this parameter-passing mechanism uses calls of the form $f(x)$ where the result ends up in x. It also means that calls of the form $f(e)$ where e is not a variable are either forbidden, or, if allowed, any result is lost. In this example here, the function is called with an array-access as argument, in which case the result is not lost, but is expected to be stored back into the adequate array slot.

Now to the example: since the call-parameters are passed by value, x and y in the body *start out* with the *values* $(1, 1)$, no aliasing here, since x and y are different variables. Both are then increased independently in the body, resulting in $(2, 2)$, and i is increased as well, giving 1. At the end of the procedure, the results in x and y are stored back to their origin.

One problem mentioned in the lecture is: if both are stored back, then: *in which order* is that done. In the situation as here, as the arguments are aliases, both return values will be stored back *into the same* memory location, and only one can "win", the last one which is written is "the" return value. In this particular example here, this problem is actually not really visible, as x and y happen to have the same value at the end of the procedure, but that's just by happenstance.

The second problem was not directly mentioned in the lecture, and that has to do with the specific fact that the arguments are not just aliases, but the actual parameters are array*expressions* $a[i]$. Upon return, the return value(s) are handed back to to the address of $a[i]$ (which can be determined by address calculation: base address of a + i times some scale factor). However, *i has changed during the call.* The question therefore is, whether to take the address of $a[i]$ *at call-time* or *at return-time.* Depending on that choice, the result is either $(2, 1)$ (address of $a[i]$ at call time) or $(1, 2)$ (address of $a[i]$ at exit).

These unclarities make hopefully clear why that mechanism is no longer considered a dignified way of parameter handling and that it has fallen from grace for good reasons.

Call-by-name: On the one hand, the situation for call-by-name can get equally complex or confusing than the call-by-value-result. That's despite the fact that the "conceptual explanation" of call-by-name is rather straightforward, namely: the parameters are passed by simple *textual substitution*[6] If we do that for the procedure, the body expands to

```
    a[i] = a[i] + 1;
      i  = i + 1;
    a[i] = a[i] + 1
```

i is globally declared and has a value of 0 at the beginning.[7] So we increase first $a[0]$ in a first step, and afterwards $a[1]$, resulting in $(2, 2)$.

The results are summarized in Table 1. □

[6] That may be a simple *explanation* of parameter passing, but still it might be complex to implement and might lead to counter-intuitive or confusing behavior.

[7] While textual substitution is the parameter passing mechanism here, still the rules of lexical/static binding apply. So in general we would have to face the problem to find out where a variable such as i here has been declared. This particular "complication" does not really apply here, as there is only one global declaration of i, so no confusion (via dynamic binding or "variable capture") is possible here.

by-value	by-reference	by value-result	by-name
$(1,1)$	$(3,1)$	$(2,1)$ / $(1,2)$	$(2,2)$

Table 1: Summary of results

Exercise 6 (Parameter passing) Give the output of the following program (written in C syntax) using the 4 parameter passing methods discussed in the lecture.

```
#include <stdio.h>
int i = 0;

void swap (int x, int y)
{
  x = x + y;
  y = x − y;
  x = x − y;
}

main ()
{ int a[3] = {1,2,0};
  swap(i,a[i]);
  printf(”%d␣%d␣%d␣%d\n”,i,a[0], a[1],a[2]);
  return 0;
}
```

Solution: Here, we keep the answer shorter, for the concepts and further explanations, see the previous exercise (and/or the lecture/book). The body of the function is intended to "swap" integer values in a fancy way, by doing some calculations (and assuming no *MAXINT* overflow happens . . .). The *actual* parameters are an integer variable (which is non-reference data) and $a[i]$ which is a non-reference integer as well, but of course a itself is a reference data type and determining $a[i]$ involves address caluclation depending on the value of i (as in the previous exercise).

Call-by-value: Call-by-value has no side-effect. The body swaps the values for x and y, but that has no effect on the actual parameters.

Call-by-reference: No *aliasing* this time! Thus, the swapping works as intended on the actual parameters, which means they are swapped.

Call-by-value-return: As there is no aliasing, in principle that makes also that easier. However, one argument is i, which is used as index for $a[i]$ which the body of the function *changes*. Consequently, the same problem as in the previous exercise needs to be considered: *when* to determine the *address* $a[i]$ for passing back the value: upon entry or upon exit of the function.

When doing it upon entry, one avoids a "re-indexing" when storing back the result and under this (plausible) assumption call-by-value-result behaves like call-by-reference (since no aliasing).

Actually, also when doing it upon exit, it may seem that it's ok, because the body of the function has not changed anything. But still bad things may happen: note that upon return, we are storing back the results in a particular order (more "dignified" would be actually to store them back *at once*). One plausible way of storing the results back is: from left-to-right wrt. the formal parameters. That means: when storing back the *second* parameter, the first one i *has already received its returned value*, which is 1 in the example. So for the second formal parameter, the execution of the body and the “half-way return” of the first result *did indeed change the* i.

Therefore: if *now* is the time to determine the address of $a[i]$, right before actually doing it (and not perhaps before seen at all variables at once), the result is stored not back to $a[0]$ but at $a[1]$. It’s a truly **awful** way of parameter passing …

Call-by-name: Again, substitution is easily done

```
  i    = i + a[i];   // i = 1
 a[i]  = i − a[i];   // a[1] = 1 − a[1] =  1−2 = −1
   i   = i − a[i];   // i    = 1 − (−1) = 2
```

Basically, mutable data and here, especially “indexed data”, where the values are accessed *simply don’t work together with call by name at all*, the result can be thorougly confusing. That’s why Louden claims that call-by-name is not used anymore. It should be noted that declarative languages, like Haskell (or Miranda), which shy avay from side effects more or less completely (“purely” functional languages), they put call-by-name to good use (the variant there is called lazy-evaluation, which is an optimization of call by name, but the mechanism is the same). One can generally say: a substitution-based explanation of parameter passing works *only* in functional setting.[8]

The results are summarized in Table 2.

by-value	by-reference	by value-result	by-name
$(0,[1,2,0])$	$(1,[0,2,0])$	$(1,[0,2,0])$ / $(1,[1,0,0])$	$(2,[1,-1,0])$

Table 2: Summary of results for i and 3 elements of a

[8] As mentioned shortly before: the actual implementation if such a language is typically not directly based on “textual substitution” of, for instance, ASTs.