# Course Script

## INF 5110: Compiler construction

INF5110, spring 2022

Martin Steffen

# Contents

# Chapter 8 Run-time environments

What is it about?

**Learning Targets of this Chapter**

1. memory management
2. run-time environment
3. run-time stack
4. stack frames and their layout
5. heap

**Contents**

## 8.1 Intro

The chapter covers different aspects of the run-time environment of a language. The RTE refers to the design, organization, and implementation of how to arrange the memory and how to access it at run-time. One way to understand the purpose of RTEs is: they have to maintain the *abstractions* offered by the implemented programming language.

More concretely: The programming language speaks about variables and **scopes,** but ultimately, when running, the data is arranged in words or sequences of bits, somewhere in the memory, and the data must be addresseed adequatly. “Abstractions” that need to be taken care of (i.e., code must be generated for that) include variables inside scopes, static and dynamic memory allocation, parameter passing, garbage collection. The most important control abstraction in languages is that of a “**procedure**”. Connected to that is the run-time **stack**.

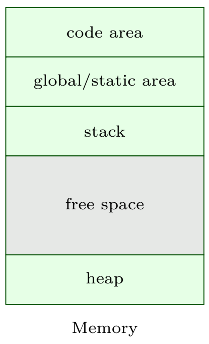


Figure 8.1: Typical memory layout

Figure 8.1 represents schematically a typical layout of the memory associated with one (single-threaded) program under execution.

One general division is that of **static** vs. **dynamic** memory. Static means, addresses of items placed in that part are known at compile time, and dynamic means, they are not.

The static memory contains data like global variables, but also the code of procedures or functions are typically allocated in the static part of the memory. The dynamic part of the memory consists of a stack and of a heap, typically.

At the highest level, there is a separation between "control" and "data" of the program. The "control" of a program is program code itself; in compiled form, of course, the machine code. The rest is the "data" the code operates on. Often, a strict separation between the two parts is enforced, even with the help of the hardware and/or the operating system. In principle, of course, the machine code is ultimately also "just bits", so conceptually the running program could modify the code section as well, leading to "self-modifying" code. That's seen as a no-no, and, as said, measures are taken that this does not happen. The generated code is not only kept immutable, it's also treated mostly as static (for instance as indicated in the picture): the compiler generates the code, decides on how to arrange the different parts of the code, i.e. decides which code for which function comes where. Typically, as indicated at the picture, all code is grouped together into one big adjacent block of memory, which is called the **code area**.

The above discussion about the code area mentions that the control part of a program is structured into *procedures* (or functions, methods, subroutines . . . , generally one may use the term *callable unit*). That's a reminder that perhaps the single most important abstraction (as far as the control-flow goes) of all but the lowest level languages is function abstraction: the ability to build "callable units" that can be reused at various points in a program, in different contexts, and with different arguments. Of course they may be reused not just by various points in one compiled program, but by different programs (maybe even at the same time, in a multi-process environment). A collection of such callable units, arranged coherently and in a proper manner (and together with corresponding data structures) is, of course, a *library*.

The static placement of callable units into the code segment is not all that needs to be arranged. At *run-time*, making use of a procedure means *calling it* and, when the procedure's code has executed till completion, *returning from it*. Returing means that that control continues at the point where the call originated (maybe not exactly at that point, but "immediately afterwards"). This call-and-return behavior is at the core of realizing the procedure abstraction. Calling a procedure can be seen as a jump (`JMP`) and likewise the return is nothing else than executing an according jump instruction. Executing a jump does nothing else than setting the so-called program pointer to the address given as argument of the instruction (which in the typical arrangement from the picture is supposed to be an address in the code segment). Jumps are therefore rather simple things, in particular, they are unaware of the intended call-return discipline. As a side remark: the platform may offer variations of the plain jump instruction (like `jump-to-subroutine` and `return-from-subroutine`, `JTS` and `RTS` or similar). That offers more "functionality" which helps realizing the call-return discipline of procedures, but ulitmately, they are nothing else than a slightly fancier form of jumps, and the basic story remains: on top of hardware-supported jumps, one has to arrange steps that, at run-time, realize the call and return behavior.

That needs to involve the data area of the memory (since the code area is immutable). To the very least: a return from a procedure needs to know *where to return to* (since it's just a jump). So, when calling a function, the run-time system must arrange to remember where to return to (and then, when the time comes to actually return, look up that return address and us it for the jump back). In general, in all but the simplest or oldest languages, calls can be *nested*, i.e., a function being called can in turn call another function. In that nested situation procedures are executed *LIFO* fashion: the procedure called last is returned from first. That means, we need to arrange the remembered return addresses, one for each procedure call, in the form of a **stack**. The run-time stack is one key ingredients of the run-time system for many languages. It's part of the *dynamic* portion of the data memory and separate in the picture from the other dynamic memory part, the heap, from a gulf of unused memory. In such an arrangement, the stack could grow "from above" and the heap "from below" (other arrangements are of course possible, for instance not having heap and stack compete for the same dynamic space, but each one living with an upper bound of their own).

So far we have discussed only the bare bones of the run-time environment to realize the procedure abstraction (the heap may be discussed later): in all but the very simplest settings, we need to arrange to maintain a stack for return addresses and manipulate the stack properly at run-time. If we had a trivial language, where function calls cannot be nested, we could do without a stack (or have a stack of maximal length 1, which is not much of a stack). In a setting without recursion (which we discuss also later), also similar simplifications are possible, and one could do without an official stack (though the call/return would still be executed under LIFO discipline, of course).

But besides those bare-bones return-address stack, the procedure abstraction has more to offer to the programmer than arranging a call/return execution of the control. What has been left out of the picture, which concentrated on the control so far, is the treatment of *data*, in particular **procedure local data**, so the question is related to how to realize at run-time the scoping rules that govern local data in the face of procedure calls. Related to that is the issue procedure parameters and **parameter passing**. A procedure may have its own local data, but also receives data upon being called as arguments. Indeed, the real power of the procedure abstraction does not just rely on code (control) being available for repeated exection, it owes its power on equal parts that it can be executed variously on different *arguments*. Just relying on global variables and the fact that calling a function in different contexts or situations will give the procedure different states for some global values provides flexibility, but it's an undignified attempt to achieve something like *parameter passing*. All modern languages support syntax that allows the user to be explicit about what is considered the input of a procedures, its formal parameters. And again, arrangements have to be made such that, at run-time the parameter passing is done properly. We will discuss different parameter-passing mechanisms later (the main being call-by-value, call-by-reference, and call-by-name, as well as some bastard scheme of lesser importance). Furthermore, when calling a procedure, the body may contain variables which are *not* local, but refer to variables defined and given values *outside* of the procedure (and without officially being passed as parameter). Also that needs to be arranged, and the arrangement varies deping on the scoping rules of the language (static vs. dynamic binding).

Anyway, the upshot of all of this is: we need a stack that contains *more* than just the return addresses, proper information pertaining to various aspects of *data* are needed as well. As a consequence, the single slots in the run-time stack become more complex; they are known as *activation record* (since the call of a procedure is also known as its activation).

The chapter will discuss different indgredients and variations of activation records, depending on features of the language.

Figure 8.2 shows the general impression of the code area or code segment. It is almost always neither moved nor changed at runtime and **statically** allocated, i.e., memory content representing machine code is not moved around nor changed or newly allocated at runtime.

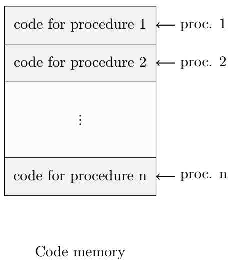


Figure 8.2: Code area

The compiler is aware of all addresses of “chunks” of code, which are the *entry points* of the procedures. The generated code is often *relocatable* and final, absolute adresses given by *linker/loader*.

The layout of the code segment here assumes that the addresses of the procedures are fixed and arranged statically in the code segment. That’s very plausible. Note that it’s not the same as fixing the question “procedure `P` occuring in the source code is located at such and such address”. That has to do with the fact that the name `P` may refer to different procedures, all under the same name. A well-known example of that is *late binding* or *dynamic binding* of methods in object-oriented languages. Binding generally refers to the association of names with “entities”, like values or procedures. That’s a central aspect of run-time environments. Sometimes, the binding can be established statically, at compile time, or dynamically, at run-time. The act of resolving the location of particular method of function, respectively jumping to that address, is also known as *dispatch*. In case of dynamically or late-bound methods, it’s called not surprisingly *dynamic dispatch*.

The phenomenon of static vs. dynamic binding is not restricted to method or function names. It can apply also to variables occuring in scopes. When talking about procedures, it’s not only methods for which dynamic binding is common. Also in languages with function variables, the dispatch has to be dynamic. That includes languages, which can take functions as arguments, in particular functional languages.

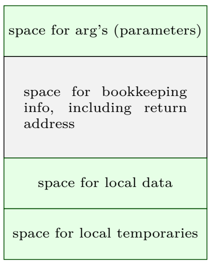

space for arg's (parameters)
space for bookkeeping info, including return address
space for local data
space for local temporaries

Figure 8.3: Schematic activation record

We will come back later to discuss possible designs for activation records in more detail, in the section about stack-based run-time environments. Activiation records (also known as stack frames) are the elementary slots of call stacks, a central way to organize the dynamic memory for languages with (recursive) procedures. There are also limitations of stack-based organizations, which we also touch upon.

## 8.2 The procedure abstraction: different layouts

In the following, we cover different layouts focusing first on the memory need in connection with *procedures* (their local memory needs and other information to be maintained at run-time, to "make it work"). Mostly, that will be a stack-arrangement, though at the end we will discuss limitations of a pure stack-based run-time environment design for function calls, and how to get memory layout more flexible than a stack arrangement.

### 8.2.1 Full static layout

A full static layout of the run-time environment means, that the location of "everything" is known and fixed at compile time. All addresses of all of the memory is known to the compiler, for the executable code, all variables, and all forms of auxiliary data (for instance big constants in the program, e.g., string literals). Such a layout is schown schematically in Figure 8.4. A fully static scheme is rarely the case for today's languages, but was the case for instance in old versions of Fortran (Fortran77). Nowayday, there could be special applications, where static layout is used, like safety critical embedded systems.

Let's look at a more concrete example in some variant of Fortan in Listing 8.1.

```
      PROGRAM TEST
      COMMON MAXSIZE
      INTEGER MAXSIZE
      REAL TABLE(10),TEMP
      MAXSIZE = 10
      READ *, TABLE(1),TABLE(2),TABLE(3)
      CALL QUADMEAN(TABLE,3,TEMP)
      PRINT *,TEMP
      END

      SUBROUTINE QUADMEAN(A,SIZE,QMEAN)
      COMMON MAXSIZE
```

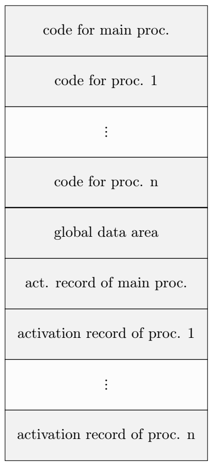

code for main proc.
code for proc. 1
⋮
code for proc. n
global data area
act. record of main proc.
activation record of proc. 1
⋮
activation record of proc. n

Figure 8.4: Full static layout

```
      INTEGERMAXSIZE, SIZE
      REAL A(SIZE),QMEAN, TEMP
      INTEGER K
      TEMP = 0.0
      IF ((SIZE.GT.MAXSIZE).OR.(SIZE.LT.1)) GOTO 99
      DO 10 K = 1, SIZE
         TEMP = TEMP + A(K)*A(K)
10    CONTINUE
99    QMEAN = SQRT(TEMP/SIZE)
      RETURN
      END
```

Listing 8.1: A Fortran example

The details of the syntax and the exact way the program runs are not so important. Also the exact details of the layout from Figure 8.5 does not matter too much.

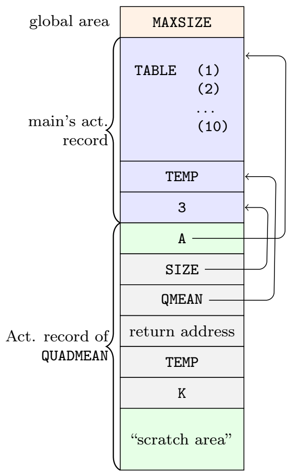


Figure 8.5: Static layout for the Fortran example

Important is the discinction between **global** variables and **local** ones, here for those for the “subroutine” (procedure). The local part of the memory for the procedure is a first taste of an **activation record**. Later they will be organized in a stack, and then they are also called *stack frames* but it’s the same thing. It’s space that is used (at run-time) to fill the memory needs when calling the function (which is also known as “activation” of the function). That needed space involves slots used to pass arguments (parameter passing) and space for local variables. Needed also is a slot where to save the return address. We said 100% exact details don’t matter, they also may depend on the platform and the OS. But what is often typical (and will also be typical in the lecture) is that the parameters are stored in slots *before* the return address and the local variables afterwards. In a way, it’s a design choice, not a logical necessity, but it’s common (also later in this chapter). It’s often arranged like that, for reasons of efficiency. Later, the layout of the activation records will need some refinement, i.e., there will be more than the mentioned information (parameters, local variables, return address) to be stored, when we have to deal with recursion.

The back-arrows in the figure refer to parameter passing and the distinction between formal and actual parameter. We come to parameter passing later.

### 8.2.2 Stack-based runtime environments

So far, the run-time environment, being static, was for languages without recursion(!), where everything is static, includingt the placement of activation records. That’s a pretty *ancient* and *restrictive* arrangement of the run-time environment.

Now, we are covering *stack-based* run-time environments, where the **stack of activation records** is part of the *dynamic* portion of the memory. The purpose of the arrangement is to offer the programmer **procedures** as **abstractions** with **local data**, including formal parameters. This leads to an run-time memory arrangement where procedure-local data together with other info (arrange proper returns, parameter passing) is organized on a stack. The stack is also called *call stack* or *runtime stack*. As always, the exact format of the activation records depends on language and platform.

Practically all full-scale programming languagues allow recursion, and therefore needs some form of dynamic management of the activation records. Many use a stack-based arrangement for managing the memory needs for procedure calls at run-time. However, the rules how and where procedures can be defined and what one can “do” with them is different in different language. In the following we discuss a sequence of complications, where the language features concerning procedures in connection with scoping requires increasing complications in the design of the activation record. The schematic structure of an activation record has been shown in Figure 8.3 earlier. The basically always have space for passing arguments, for local variables and so-called temporaries. The gray part in Figure 8.3 was said to be for book-keeping, administring necessary information to maintain the run-time stack and realize the run-time abstraction of lexical scopes. That gray part is the part of the activation record that gets more or less complex depending on the language.

Indeed, in the most complex situation, with higher-order functions, not only the activation records needs more bookkeeping information than in simpler languages, but even the stack-discipline is not longer adequate.

We mentioned that the complications reflect the treatment of the memory needs for procedures and the corresponding *scope*. The following discussion concerning the form of activation records concentrates solely on languages with *static* binding (which is more harder to achive), not dynamic.

maybe add abstraction and static variables.

**Remark 8.2.1** (Stack-based run-time environment)**.** We present stack-based run-time environments as "evolution" of a fully static environments, in particular the dynamic placement of activation records is needed to deal with recursion.

Of course, also in the restricted setting of a static RTE, calls and returns follow at runtime a LIFO discipline and in that sense, at run-time, there *is* a *stack* which conceptually grows and shrinks, but in absence of recursion, each procedure is activated at most once at each point in time which allows a static placement of the activation records. □

## C-like languages, i.e., languages without local procedures

The first complication is languages with **recursion** but all **procedures are global**. I.e., it's not possible to define a procedure nested locally inside another procedure. This is sometimes called "C-like" languages, because C is a prominent example for that situation.

One step further, in the following section, will be not surprisingly to generalize that to languages that do support nested procedure declarations (in a setting with lexically bound variables; Pascal being one example.) That's more general, and that nesting will require to introduce, besides dynamic links, also static links

The specific information needed are the **frame pointer**, the **control link** (or **dynamic link**)[a] and the **return address**. We also discuss and show in the picture the so-called, **stack pointer**.

[a]Later, we'll encounter also *static links* (aka *access* links).

The stack pointer need not to be *stored* in the activation record, but it will be part of the discussion.

**Remark 8.2.2** (Static link)**.** The notion of static links mentioned in the footnote is basically the same we encountered before, when discussing the design of **symbol tables,** in particular how to arrange symbol tables properly for nested blocks and lexcical binding. Here (resp. shortly later down the road), the static links serve the same purpose, only not linking up (parts of a) symbol table, but activation records. □

Let's illustrate the different pointer or links in a small example (see Listing 8.2). Not that it's the focus of the example, but the C-code represents a simple recursive implementation for calculating the greatest common divisor of two integers (making use of some modulo calculation in the recursive call, that's the `%` operator). C is also uses call-by-value a parameter passing mechanism. We will cover parameter passing later.

```
#include <stdio.h>

int x,y;

int gcd (int u, int v)
{ if (v==0) return u;
    else return gcd(v,u % v);
}

int main ()
{ scanf("%d%d",&x,&y);
  printf("%d\n",gcd(x,y));
  return 0;
}
```

Listing 8.2: Euclid's recursive gcd algo

A snapshot of the memory, in particular of the stack the activations of the gcd-procedure is shown in Figure 8.6. The three activation records of gcd are shown in blue. The need for remembering the *return address* should be obvious. Actually, to remember where to return to is needed also in the static layout, without recursion (but procedure calls and returns). The return address points to some place in the *code area*, which is not shown in the picture; the picture shows only the part of the memory containing data.

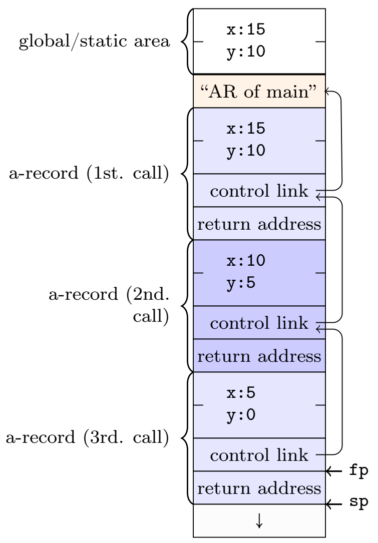


Figure 8.6: Stack gcd

- **control link**
  - aka: dynamic link
  - refers to caller's FP
- **frame pointer** FP
  - points to a fixed location in the current a-record
- **stack pointer** (SP)
  - border of current stack and unused memory
- **return address**: program-address of call-site

The picture illustrates the notion control links, the frame pointer and the stack pointer. It also shows that each of the 3 *activations* of the `gcd` procedure has its own data area where the current values of local variables are held. In this example, the only local variables of the `gcd` procedure are the formal parameters `u` and `v` (In the figures, the slots are called `x` and `y` which is correct only for the global area, for activations of `gcd`, it should be `u` and `v`).

There can be more local variables: C allows to introduce local variables, besides the formal parameters, in functions or procedures. What is not allowed is to introduce local procedures. There is another general reason, a activation record needs memory, that's for holding intermediate results when dealing with compound expressions. For that, the compiler will typically use so-called *temporary variables*, variables introduced in the code generation phase for exactly that purpose: hold intermediate results. We will see examples of that later.

The exact design of activation record may vary. The need for a concept like the **control link** comes from a simple fact, and it has nothing to do with scoping or lexical binding. It's simply one possible way to make a stack with *variable-sized slots* work. [maybe more]

The **frame pointer** points to the current activation record, i.e., the top-most entry in the stack. Note that activation records or stack frame is *not* of fixed size. In the example, with only one function, of course all activation records are of the same size. What is important that the frame pointer points to a "definite" position inside the activation record in such a way, that the local data (variables etc) can be accessed uniformely and fast. The dynamic link or control link corresponds to the frame pointer. The stack-pointer is something else, it simply demarkates the border between the stack-occupied part of the memory and the free part.

**Side remark 8.2.3** (Tail recursion)**.** As a side remark; the GCD procedure is recursive, all right. However, it makes use of a restricted form of recursion, namely **tail recursion**. In the body of `gcd`, in each branch, gcd is either not called at all, or it is called at the end of the procedure body, as last thing before returning. That's tail recursion.

It's a simply form of recursion, also in connection with run-time environments. The call which pushes a new stack frame to the run-time stack, is the last thing that happens in an activation. That means, the space of the caller's stack frame and the local data it contains therein, is not actually needed any longer. That means, one could arrange the run-time environment in such a way, not to add another stack frame for the callee, but to recycle the space of the caller's frame. If one (resp. the run-time system) really makes use of recursion, one still need to maintain a stack of return addresses, of course. However, a tail recursive situation can be completely be replaced by an iterative one, using a loop instead. A compiler that does that automatically, replacing (sometimes) recursion by iteration when possible, is said to do tail recursion optimization.

At the level of the run-time system, at machine code level (and potentially intermediate code level), there are typically no looping constructs, of course, so making use of looping instead of recursion is more a conceptual statement. Recursion would involve jumps plus arranging a stack with return addresses, so one jumps repeatedly to the beginning of a body, but at the end, one jumps back (which corresponds to a return). An iterative solution would not use a stack, and would simply loop thought the body, without need of returning; except of course, a return to the code calling from the outside needs to be done, in the example the return to the `main` method. □

The code is artificial, it will later be used to illustrate the run-time stack in a simple setting. Being called with 2 initially, there are only three activations of the 2 functions *f* and *g* altogether.

**Activation trees** An activation of a function or procedure corresponds to the *call* of thaat function. An activation record is a data structure for run-time system to hold all relevant data for a function call and the control-info in "standardized" form. It must assure the well-known control-behavior of functions, the LIFO treatment of calls and return. If data **cannot outlive** the activation of a function, the activation records can be arranged in as **stack** (like here). In this case: activation record are also called **stack frames.**

ms Error: move

**ms Error: Perhaps move elsewhere, duplicate explanation.**

Next we shortly discuss shortly the related concept of **activation tree**. While activation records and stack frames are concrete data structures that need to be realized by the compiler (writer), activation trees are a *conceptual* description what happens at run-time, which function calls which other ones.

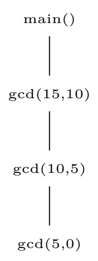
*Example* 8.2.4 (Activation records)*.* The code of Listing 8.3 contains some artificial code, which will use to illustrate the concept of activation trees (and a bit later also for another illustration of activation records and stack frames.

```
int x   = 2; /* glob. var */
void g(int);/* prototype */

void f(int n)
  { static int x = 1;
    g(n);
    x--;
  }

void g(int m)
  { int y = m-1;
    if (y > 0)
      { f(y);
        x--;
       g(y);
      }
  }

int  main ()
  { g(x);
    return 0;
  }
```

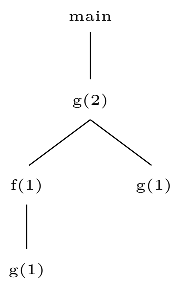
Listing 8.3: Another example illustrating scoping and activations (in C)

The code contains *local* and *global* variables under lexical, static scoping in C. On the global level, there is only one global variable namely `x`, all others are local; the formal parameters of the functions count among the local variables. Note that the procedure `f` has a local variable likewise called `x`. Finally, C is *call-by-value*, like Java, and many other languages.[1]

[1]For participants of IN2040: call-by-value, which is also the standard order of evaluation of Scheme, is there also called *applicative order evaluation.* The terminology of applicative order vs- normal order evaluation is often used in the context of functional and declarative languages. We will talk aboiut parameter passing and various evaluation strategies later.



(a) `gcd`



(b) `f` and `g` example

Figure 8.7: Activation trees

The two pictures illustrate the notion of *activation tree*, in Figure 8.7a the tree for calling the funtions of the gcd-example from Listing 8.2 on $(15, 10)$, and Figure 8.7b is the activation tree for the code from Listing 8.3.

For the gcd-example, it's not much of a tree as it's linear. An activation of gcd calls itself at most once, and actually, gcd is *tail-recursive.* □

**Variable access and layout of activation records** Activation records are structurally *uniform* per language, or at least per least per compiler) and platform. However, the activation records are not identical. In particular, activation records for different functions are of **different size** as different functions have different memory needs. But each activiation of the same function, of course, leads to the allocation of the same amount of memory (here on the stack).

Let's look at Figure 8.8, which is supposed to give a schematic view of the activation records for procedure `g` from Listing 8.3.

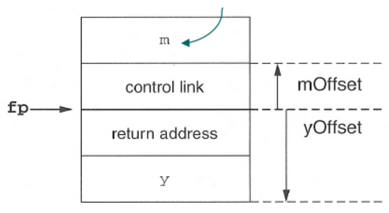


Figure 8.8: Layout `g`

It shows a plausible arrangement of `g`'s activation record (and other functions would have analogous arrangements. In the picture, `fp` is the frame pointer, `m` (in this example) is the (only) parameter of the function (here `g`).

For the stack, we assume that it grows to addresses lower in the stack space. As far as depicting the address space is concerned, we draw higher addresses "higher up" in the picture, and consequently, the stack "grows" downwards. For the pointers in such pictures: the "pointers" or arrows point to the "bottom" of the meant slot.[2] Different presentations may employ different graphical conventions. The graphical conventions are of course to be

[2]In some pictures later, we let them point also to the "middle" of a slot.

distinguished from the layout itself of the activation record and the corresponding calling conventions (see later).

In Figure 8.8 and with thes conventions the `fp` points to the control link, i.e., the memory (perhaps a specific register) corresponding to the frame pointer contains the address of the control link. To say it differently, the control link as an **offset** of 0 from where the `fp` points to. The return address given the slot below has a negative offset to that pointer.

Roughly speaking, the frame point points "to" the activation record at the top of the stack, which is the record of the current activation at a given point. However it does not point to the "top"[3] of that frame or stack, but to a well-chosen, well-defined position in the frame; in the shown layout, this well-chosen anchor point is the location of the control or dynamic link. All local data, for instance local variables are accessible *relative* to that, some with a positive **offset**, some with a negative offset, and, as mentioned, the control link is directly pointed at with the frame pointer, with an offset of 0.

As explained, the control or dynamic link is located with an offset of zero (and the return address with an offset of whatever the space need of the return address is). It's not a logical necessity to have them there, the only thing required is that they are located at a known place in the activation record pointed at by `fp`. So depending on the language, compiler, platform etc. there may be other arrangements as well.

However, doing it the way discribed is a common and plausible one. It has to with the desire to build up the activation records in an efficent and clean way. When executing a call at run-time, a new activatation record is **pushed** to the stack. But that is's not a instantaneous thing, it is a step-by-step process, filling one slot after the other, i.e., pushing one slot after the other to the stack aready; and when all the slots are filled, one can see it as having pushed the whole activation records. The steps that do that, connected to the layout of the activation records, are called a language's (or platform's) calling conventions (see later).

⇒ *frames* on the stack differently sized

**Layout for arrays of statically known size** Procedures can of course also declare locally data more complex than data of basic or elementary types. The code from Listing 8.4 show a procedure with a local array of statically known size.

```
void f(int x, char c)
{ int a[10];
  double y;
  ..
}
```

Listing 8.4: Procedure with local array

To calculate the memory need and this the offsets inside an activation record is not much more complex for arrays (see Table 8.1). As conventional, the identifier `a` represents also the (address of the) first slot of the array `a(0)`.

[3]Confusingly, the top of the stack is written at the bottom.

| name | offset |
|---|---|
| x | +5 |
| c | +4 |
| a | -24 |
| y | -32 |

Table 8.1: Array of fixed size: offsets

The layout for a corresponding activation is depicted in Figure 8.9 and the codes from Listings 8.5 and 8.6 how the variables and the array content can be **accessed,** given the frame pointer.

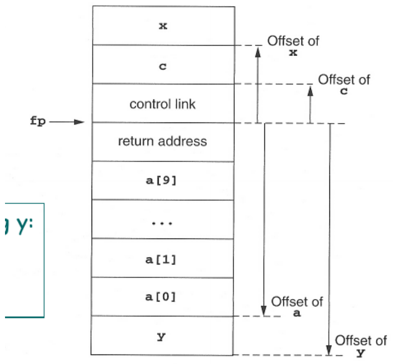


Figure 8.9: Layout

```
c:  4(fp)
y:  −32(fp)
```

Listing 8.5: access of `c` and `y`

```
(−24+2*i)(fp)
```

Listing 8.6: access for `a[i]`

The example makes some not unplausible assumptions on the size of the involved data. The addresses count 4 words, the character 1, the integers 2 words, the double 8. Notation like `4(fp)` is meant as syntax to designate the memory interpreting the content of `fp` as address and add 4 words to it. We will later encounter, in the context of (intermediate) code, different addressing modes (like indirect addresses etc). Except in very early times, hardware gives support for more complex ways of accessing the memory, like support for specifying given offsets.

### Calling conventions: pushing and popping activation records

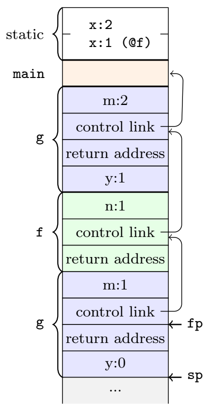
The call stack is a stack of activation records; calling a proceduring involves pushing an activation record and returning means popping off an activation record. This is a view on the stack on the “macro”-level, so to say, which activation records as elements of the stack, and the **frame-pointer** pointing the top-most record, resp. pointing to a well-defined anchor point inside that top-most record. But the stack as well as the pushing and popping has also some “micro”-level. The stack consists of individual words, and the top of the stack, separating the occupied part of the memory from the free one, is pointed at by the **stack pointer**. So pushing a new activation record onto the stack involves, at the micro-level, to copy the relevant information step by step into the stack, placing them at the designated places inside the activation record.

Someone has to perform those steps; they have to be executed each time a function is called and "undone" when returning from a call. Since that happens at run-time, one cannot say that those steps are *executed by* the compiler. But the compiler resp. compiler writer has to arrange for that they are done executed at run-time. Concretely, the code generator has ultimately to inject small corresponding stretches of instructions, that perform the necessary steps at machine code level.

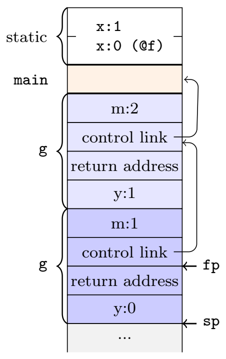
Same as the the activation record design is uniform, to the very least per compiler, the steps that push and pop the activation records are *uniform*, i.e., done in the same order for each call (corresponding to a push) and for each return (corresponding to a pop).

Those the resp. the specification of thos steps are called **calling sequences**, or also **linking conventions** or **calling conventions**.

*Example* 8.2.5. Back to the C code again from Listing 8.3 with the functions $f$ and $g$.



(a)



(b)

Figure 8.10: 2 snapshots of the call stack

Figure 8.10 shows to snapshots of the stack. The stack from 8.10a shows 4 activation records. It corresponds to the point at run-time, whe $g$ is called for the second time, but the execution has not yet returned from the activation. See also the activation tree from Figure 8.7b. The stack-pointer separates the used part of the stack from they free one (in grey).

In the used part of the stack, thre are 4 activation records, one for the main-function, and 3 for the 2 activations of $g$ and the single activation of $f$.

Besides the stack, which is part of the dynamic memory, also part of the static memory is shown (in white). The example program contains a global variable $x$. Additionally, the function $f$ likewise has a variable called $x$, which is declared as `static`. As a consequence, *that* $x$ is also stored in the static part of the memory (marked as `@f` in the picture). The second picture shows the sitation after the call to $f$, and $g$ has again be called. That corresponds to the right-most branch of the activation tree from Figure 8.7b.
□

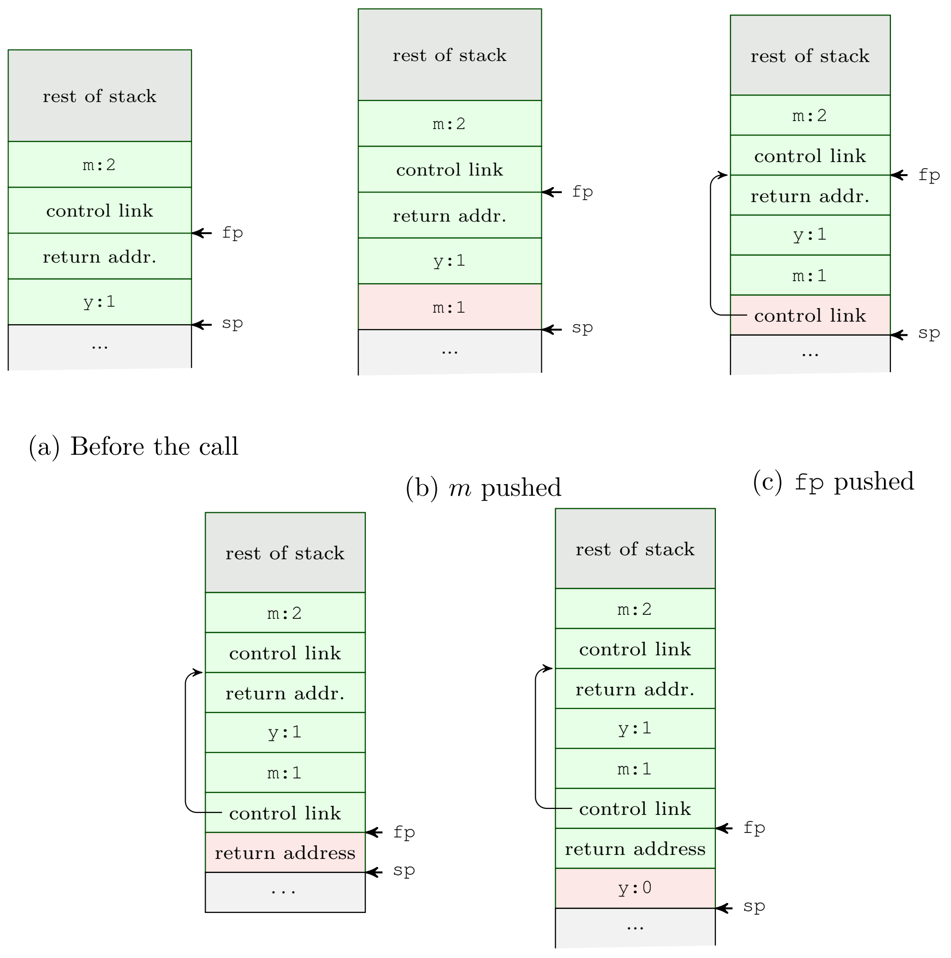
- For procedure call (entry)
  1. compute arguments, store them in the correct positions in the *new* activation record of the procedure (pushing them in order onto the runtime stack will achieve this)
  2. store (push) the `fp` as the *control link* in the new activation record
  3. change the `fp`, so that it points to the beginning of the new activation record. If there is an `sp`, copying the `sp` into the `fp` at this point will achieve this.
  4. store the return address in the new activation record, if necessary
  5. perform a *jump* to the code of the called procedure.
  6. *Allocate space* on the stack for local var's by appropriate adjustement of the `sp`
- procedure exit
  1. copy the `fp` to the `sp` (inverting 3. of the entry)
  2. load the control link to the `fp`
  3. perform a jump to the return address
  4. change the `sp` to pop the arg's

*Example* 8.2.6 (Calling sequence). Let's look one more time the C code again from Listing 8.3 with the functions $f$ and $g$. The steps when calling $g$ are shown in the pictures from Figure 8.11. □

**Treatment of auxiliary results: "temporaries"** As explained, activation records contain space for various kinds of data, including space for local variables, including values for the formal parameters of the function in question. There is one part of the local memory need that we have not yet mentioned. The code of the method body will in general do some calculation. At souce code level resp. in abstract syntax, that involves *compound expressions*. Such expressivity will not be available in the machine code and typically neither in intermediate code. There are different forms and flavors of Intermediate code, but typically intermediate code is platform independent and somewhere "half way" between souce code and the ultimate assembler code. At any rate, also typical intemediate code does not support compound expressions. Instead, intermediate code breaks down compound calculation into their basic steps uses additional local variables to store intermediate results. Those are known as temporary variables, or **temporaries**. It's one task of intermediate code generation to introduce those additional variables, i.e., generate intermedate code making use of those. Like procedure-local variables introduced at source code level, also intermediate variables need to be stored in the activation records as well, of course. Actually, on the (intermediate) code level, there is no real difference between "official" local variables and temporary variables. Both represent values stored there, and ultimate slots i.e., addresses arranged within the activation record. Intermediate code and temporaries will be discussed in a later chapter.[4]

For concreteness' sake, let's look at the C-code snippet from Listing 8.7.

[4]We will look later at two flavors of intermediate code, only one will actually make use of temporaries (three-address code), the other one will manage intermediate results in a different way.



(a) Before the call

(b) $m$ pushed

(c) `fp` pushed

(d) `fp := sp`, push return addr.

(e) alloc. local var `y`

Figure 8.11: Steps when calling $g$

```
x[i] = (i + j) * (i/k + f(j));
```

Listing 8.7: Compound expression

The computations in the example are not really complex from programming perspective, but they are *compound*. Perhaps the hardware (and the intermediate code) has support for `x + y`, `x-y`, `x+1` etc., but compound expressions like the one in the example are of course not natively supported. They have to be broken down to elementary calculations and the intermediate results need to be stored somewhere, in *temporaries* and the activation record must provide enough space so be able to locally store those results.

**Variable-length data** The examples so far involved variables containing fixed-sized data, including the temporaries. Next we shortly discuss variable-length data, concretely corresponding arrays. Ada is a language that supports that data structure.

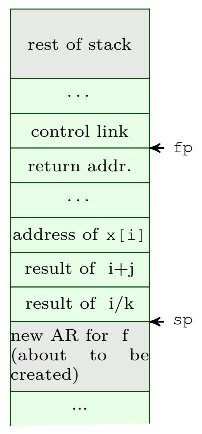


Figure 8.12: Temporaries

```
type Int_Vector is array(INTEGER range <>) of INTEGER;

procedure Sum(low,high: INTEGER; A: Int_Vector) return INTEGER
is
  i: integer
begin
    ...
end Sum;
```

Listing 8.8: Variable-length arrays in Ada

The previous C example from Listing 8.12 mentioned the use of arrays already (involving `a[i]`), but that's just a slot in an array, which was otherwise not described. In particular it was not specified in the example whether the array was declared outside or locally to the procededure, resp. if it was passed as argument.

Here, we illustrate how to deal with the situation that an variable-length array is passed *as argument*. We assume that the array is passed *by value*, i.e. the callee receives a copy of the array. In many languages, including C and Java, this is not the way arrays are passed. Typically, they are of "reference type" which means, a reference to the array is handed over to the callee, i.e., copied into the activation record. But, as said, the example is how to handle the parameter passing in ea by-value context.

The treatment is actually unproblematic; Figure 8.13 shows a possible layout . The picture simply says: if an array passed as argument, the type may not specify its size, because when passing the array, the size is known. Then just store the **size** at one particular, agreed upon place in the activation record (here at offset 6), and then use the value for the calculation when accessing a slot. So, compared to the previous handling of arrays, there is just one layer of indirection involved. In the shown example, the access for `A[i]` would, for instance, be calculated as `@6(fp) + 2*i`

**Nested declarations** Before moving on to the more complex situation of nested procedure declarations, let's have a look at how do deal with blocks or scopes inside a procedure.

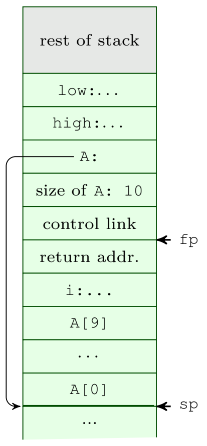


Figure 8.13: AR layout

A block or scope nested inside another block or scope, is sometimes called **compound statement** in the context of C.[5]

```
void p(int x, double y)
{ char a;
  int i;
  ...;
 A:{ double x;
    int j;
    ...;
  }
  ...;
 B: { char * a;
    int k;
    ...;
  };
  ...;
}
```

Listing 8.9: Nested declarations

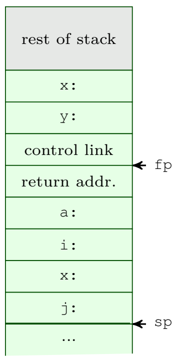
Listing 8.9 shows a simple situation, with scopes A and **B** nested inside a procedure p.

The gist of the example is: if one has local scopes of that kind "side by side" in the code, here called A and B, there is no need to allocate space for both. The space for the local variables from the first scope maybe reused for the needs of the second. There is also no need to officially "push" and "pop" activation records following the calling conventions, though nested scopes do follow a stack-discipline and they could be treated as "inlined" calls to anonymous, parameterless procedures.

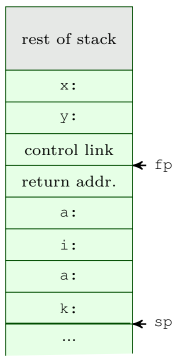
## Stack-based RTE with nested procedures

What follows in this section (illustrated with Pascal), is to relax one restriction we had so far wrt. the nature of variables. It may not have been obvious, but it should become so now: We were operating with a C-like language, which is meant as featuring lexical scoping and non-nested functions or precedures. That means: there are only two "kinds"

[5]The terminology of compound statement is also used to simply mean non-elementary statements, without that it necessarily involves a nested scope. Here, we mean the nested-declarations interpretation of the word.



(a) Area for block A allocated



(b) Area for block B allocated

Figure 8.14: Steps when calling *g*

of variables: global ones, which are static, and local ones which reside in the current stack frame.

Now, with nested procedures (and still lexical scoping) there are variables neither static nor residing the the current stack frame. So we need a way to access those during run-time. That will be done introducing **static links**.

Languages with nested procedure (but without higher-order functions) are called **Pascal-like** languages.

The code from Listing 8.10 is not just Pascal-like, it is in some concrete Pascal dialect. The comments after the `begin` and `end` statements indicate to which procedure that part belong. Since `q` is nested in `p`, and since `p` has a local variable `n` in the same scope, this local variable `n` is accessible inside `q`. At run-time, in a call to `q`, the corresponding activation record will reside on the run-time stack. If the body of `q` makes use of `n` (not explicitly shown in the skeletal code), it needs a way to locate the variable's content. From the perspective of `q`, the variable is **neither local to `q` nor global**. It's of course local to *p*

```
program nonLocalRef;
procedure p;
   var n :  integer;
   procedure q;
   begin
      (* a ref to n is now non-local, non-global *)
   end; (* q *)

   procedure r(n : integer);
   begin
      q;
   end; (* r *)

begin (* p *)
   n := 1;
   r(2);
end; (* p *)

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

Listing 8.10: Nested procedures in Pascal

Figure 8.15 shows a situation where the main program calls `p`, which calls `r` which calls `q`. Not all details of the activation records are shown in the picture, it's just meant as showing the shape of the stack. As seen in the code from Listing 8.10, procedure `q` accesses the variable `n`. Under **lexical** scoping, that refers to `n` declared in `procedure p`.

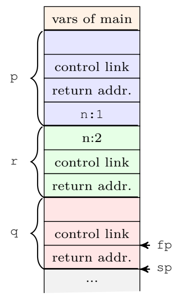


Figure 8.15: Stack frames (general) for calls m → `p` → `r` → `q`

This lexical nesting of the scopes is not the same as the call-hierarchy, i.e., the static nesting is not reflected in the *run-time* call stack, of course. Fortunately, though, the call-stack and the lexical nesting is not completely indepedent. Variables defined locally in a procedure, like `n` in `p`, can of course me accessed inside `p`, including by procedures that are defined locally in `p`, like `q` and `r` (though in the shown code, `r` does not actually access `n`).

Now, since `q` (and `r`) are defined locally *inside* `p`, no-one outside can call them, which would require to allocate an activation record for them when executing the "outsider". In other words

`q` (and `r`) can be called and activated only during times when `p` is activated, i.e., when there is at least one activation record for `p` on the stack. With `p` calling `q` (or `r`), `p`'s activation record is earlier allocated than that of the callee (and not yet de-allocated), which means it can be found deeper on the stack. The **static link** (aka **access link**) points to the most recent activation of the statically surrounding scope and that serves to access the relevant activation record. With a nesting depth of more than one, it may involve following *multiple* static links.

Figure **??** shows the same sitation as the run-time stack from Figure **??**, this time filling out more details. One crucial addition is of course, the **static link** indicated in red.

As everything else, it's placed at a well-specified position inside the activation record, with a known offset from the frame pointer. In the picture, it's shown at a *fixed* position right beside the control (or dynamic) link. It points to the stack-frame representing the current AR of the statically enclosed "procedural" scope.

The access links, same as control links point "to" a stack frame. As explained, the point of reference for a frame is not the start, not the end of the stack frame; the "anchor point" of the stack frame is the where the frame point points to, when the stack frame is on top of the frame. And that is (in the shown layout) also the slot that contains the control links.

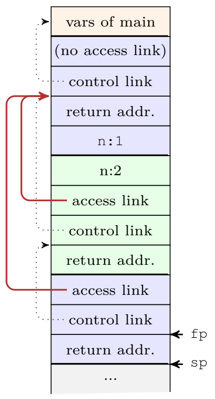


Figure 8.16: Static (or access) link

**Remark 8.2.7** (Lexical scoping and static links)**.** remember: static links (or access links) in connection with *symbol tables* **Symbol tables**

- "name-addressable" mapping
- access at compile time
- cf. scope tree

**Dynamic memory**

- "adresss-adressable" mapping
- access at run time
- stack-organized, reflecting paths in call graph
- cf. activation tree

□

As mentioned, procedures can be nested deeper than just involving one level. Listing 8.11 shows an example of that.

```
program chain;

procedure p;
var x : integer;

   procedure q;
      procedure r;
      begin
         x:=2;
         ...;
         if ... then p;
      end; (*r *)
   begin
      r;
   end; (* q *)

begin
   q;
end; (* p *)

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

Listing 8.11: Example with multiple levels

In the example, procedure `p` contains procedure `q` and that in turn contains `r`. That is the static structure, which is relevant for lexically scoped variables. At run-time, the main procure calls `p` which calls `q` which calls `r`, so that order is somehow aligned with the static nesting structure, but that's not the point of the example. It's not a complete coincidence, a call chain like `p` calls `r` which calls `q` is of course not possible, because `r` is is nested inside `q`.

When the inner prodecure `r` is called, the variable `x` is accessed. That is declared in the body of procedure `p`, **which is two static nesting-levels away from that.** To find the appropriate activation record, one needs to dereference the access link 2 time (or in general multiple time), a phenomenon called **access chaining.**

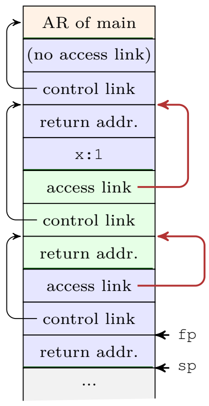


Figure 8.17: calls m → `p` → `q` → `r`

If we use dot-notation to access slots inside an actication record, we could write `fp.al.al.x` for the situation of the example. Note that in the sketched design of activation records, the static access link `al` is at a fixed offset inside AR (as is `x` in its corresponding record). Of course the activation records for different procedures are differently sized, which means the actual offset from the activation record for `r` to `x` is `statically unknown!`. What is statically known is the **number of access link dereferences** which reflects the lexical nesting situation. The task of the compiler is generate an appropriate access chain with the chain-length statically determined, but of course the actual computation is done at run-time

how can access chaining be **implemented**? Implementing means, generating (machine) code that accesses the corresponding pieces of data in the correct activation record, thus making the lexical scoping abstraction of the programming language a reality at run-time. Let's have a look at the following situation

```
fp.al.al.al.  ...  al.x
```

involving an access chain to access a variable `x`. The access, chained or not, needs to be fast, which means, one would use *registers*, resp. one would design it in such a way that the frame pointer `fp` is held in dedicated register.

```
4(fp) -> reg    // 1
4(reg) -> reg   // 2
...
4(reg) -> reg   // n = difference in nesting levels
6(reg)          // access content of x
```

Listing 8.12: Access chaining

The "machine code" plausibly uses registers to follow the change. It's assumed that the static link is contained in an offset of 4 in the activation record (pointed at via the frame pointer `fp`, which also may be kept in a dedicated register, like typically the stack pointer). Variable `x` is assumed at an offset of 6 in the frame that corresponds to the scope where `x` is defined.

Of course, following a chain of access links is costly. In practice, very long chains may not occur very often. It's at least for languages like Pascal. On the other hand, in languages where functions play a central role (i.e., in functional languages), a programmer may well structure the code with functions nested inside functions nested inside functions etc. Of course that depends a bit in the problem and the personal programming style, but still, nesting of functions comes easy in functional languages.

It should be noted that a stack-based run-time environments will no longer be doable for fully higher-order functions; we will cover that later to some extent. However, the concept of static links is still relevant then, even if it does not connect slots, i.e., activation records, on a stack.

Now, that the activation records has mildly become more complex, adding static links, also the calling sequence need to be adapted, but that's like a mild adaption, and the principles of what calling sequences do is conceptually unchanged.

**Calling sequence**, now with static links: For procedure call (at entry):

1. compute arguments, store them in the correct positions in the *new* activation record of the procedure (pushing them in order onto the runtume stack will achieve this)
2. 
   - **push access link**, value calculated via link chaining
   - store (push) the `fp` as the *control link* in the new AR
3. change `fp`, to point to the "beginning" of the new AR. If there is an `sp`, copying `sp` into `fp` at this point will achieve this.
4. store the return address in the new AR, if necessary
5. perform a jump to the code of the called procedure.
6. Allocate space on the stack for local var's by appropriate adjustement of the `sp`

For procedure exit, steps are reversed

1. copy the `fp` to the `sp`
2. load the control link to the `fp`
3. perform a jump to the return address
4. change the `sp` to pop the arg's **and the access link**

The previous example from Figure 8.17 illustrated lexical nesting involving more than just one nesting level, which required this access chainging. There is another aspect we have mentioned only in passing. The access link points to an activation record corresponding to the lexically enclosing procedure. That's fine and good, bit of course the enclosing procedure could have called multiple times in recursive situation, which means that there are **multiple activation records** for the enclosing procedure on the stack. Which is the one should the static link point to? Now when pressed for an answer, it's probably clear: it's the "most recent" stack frame. That's illustrated in Figure 8.18.

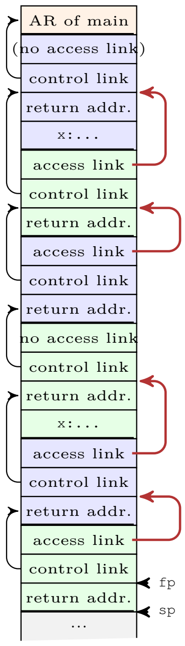


Figure 8.18: `main` → `p` → `q` → `r` → `p` → `q` → `r`

### Functions as parameters, closures, and higher order functions

There is more to scoping and run-time environments tha nested procedure declarations. We have seen glimpses of that before, mostly in the context of Pascal. In particular, in the chapter about type checking, we haveseen a Pascal example with *procedure variables*. We will revisit that example here. We also shortly mentioned in connection with static links, without a concrete example involving procedure variables and without going into details, that those variable complicate matters. Ultimately, when dealing with full higher-order functions, one cannot arrange the activation records on a stack.

Of course, even with higher-order function, the calls and returns follow a LIFO discipline. So, there is still a notion of call-stack. The stack in the run-time system is not just there to manage the return addressed and to regulate thereby the proper control-flow of calls and returns in a stack-like manner. The stack also allocates and de-allocates the memory needs of the activated functions (plus some mechanism to find the proper lexical scope, if the language is lexically scoped; that's the static links).

Anyway, that a stack arrangement for the data works for languages where the life-time of the data for a function activation is *aligned* with the life time of the activation itself: when a function returns, removing the return address for the stack, also the local data is no longer needed. This means, one can treat the return addresses and the data jointly on a stack the way dicussed.

For higher-order function, this alignement of the life-times of function activations and data declared in a function is no longer given. Therefore, one need a more general form of run-time environment, putting the activation records on the **heap**. The corresponding concept is typically not called activation record any more, but it called **closure**.

**Procedures as parameters** We start less ambitious, though, we don't fully embrace higher-order functions, but look at functions or procedures as parameters only. In that setting, the alignment of local data and function allocation *still* holds, though it get's more complex. Anyway, with this alignment one *still* can make a stack-based arrangment. In a way, one has *stack-arranged* closures. However, generally, when talking about full closures, they are normally heap arranged. There are not many languages nowadays that bother to support procedure parameters without also support procedures as return values.

Pascal does, so it's not a higher-order, but actually since Pascal supports procedure variables, there is a mechanism to "return" a function as side-effect. That means, Pascal's stack-based run-time environment design will run into trouble, as we will see.

Anyway, Listing 8.13 shows the Pascal example again, the one from the type checking chapter.

```
program closureex(output);

procedure p(procedure a);
begin
   a;
end;

procedure q;
```

```
var x : integer;
   procedure r;
   begin
      writeln(x);    // ``non-local''
   end;

begin
   x := 2;
   p (r);
end; (* q *)

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

Listing 8.13: Procedures as parameters

Listings 8.14 and 8.15 contain the "same" example in go and ocaml. To test the code, perhaps Go or ocaml compilers are more easy to get hold of. Unlike Pascal, Go and ocaml support higher-order functions, but that includes passing functions as parameters.

```
package main
import ("fmt")

var p = func (a (func () ())) {  // (unit -> unit) -> unit
        a()
}

var q = func () {
        var x = 0
        var r = func () {
        fmt.Printf(" x = %v", x)
        }
        x = 2
        p(r)    // r as argument
}


func main() {
        q();
}
```

Listing 8.14: Procedures as parameters, same example in Go

```
let p (a :unit -> unit) : unit =   a();;

let q() =
  let x: int ref  =  ref 1
  in let r = function () ->  (print_int !x) (* deref *)
  in
  x := 2;    (* assignment to ref-typed var *)
  p(r);;

q();;  (* ``body of main'' *)
```

Listing 8.15: Procedures as parameters, same example in ocaml

Here, as said, we are not "going full higher-order" and this can still stay within a stack-array arrangement of the activation records. However, we need to add another complication, which can be called **closure**. As far as closures are concerned, it will be a rather *restrictive* form. Typically, when talking about closures, those are data structure for managing the run-time needs for higher-order functions and one might stumble upon statements like "closure are heap-allocated". In general, that's the case, but, as said, we are restricting ourselves to procedures as parameters, which allows to stick to a stack-arrangement. One could therefore say, we are talking about **stack-allocated closures**. The discussion is is also rather low-level, i.e., we talk about a specific way to achieve closures, namely by a modest extension on our stack frames. Finally, the presentation here presents a specific semantics of you non-local variables are treated by the stack allocated closures, they are treated **"by reference"**. The concepts of passing values "by reference" and "by value" will be discussed in the section about parameter passing. Closure have no formal parameters, so one normally does not speak about parameter passing here, but the distinction of accessing values of variables outside the current activation either by value or by reference makes also sense for closures. As said, we only look at a by-reference design.

Before we see how to extend the design of the activation records, let's start with a **conceptual** picture of what a closure is, independent from concrete design of the run-time environment, the activation records, and of whether they are stack or heap allocated.

> A closure is a function body *together* with (access to) the values for *all* its variables, including the non-local ones.

When saying, the closure "contains" the function body, we mean in an implementation not that the code is stored in he closure, it's rather a pointer to where to find the code in the code segment (at least in the standard layout)

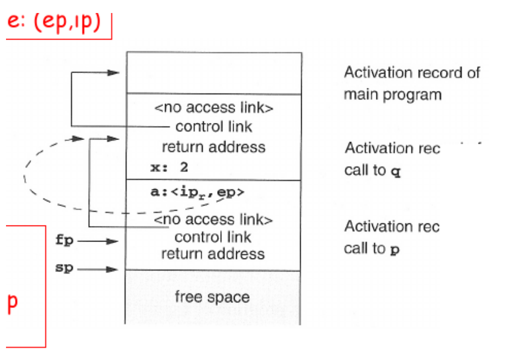
Let's go back to the Pascal example from Listing 8.13. The call chain in the example is that the main program calls `q` which in turn calls `p`, and when calling `p`, a procedure is handed over, called `r`.

When `p` is called, of course a stack frame is allocated. If `p` is executed, upon being called, it calls a procedure refered to as `a` in the body of `p`. That variable of course is just the formal parameter of the procedure `p`.What is actually called is the **actual** parameter of the call, in this example that's the procedure `r`. but that's just *parameter passing,* when calling `p`, a pointer to `r` will be handed over and store in the activation record of `p`. With (reference to) `r` stored, one can arrange the call to `r`, i.e., jump to the corresponding address in corresponding step the call-sequence.

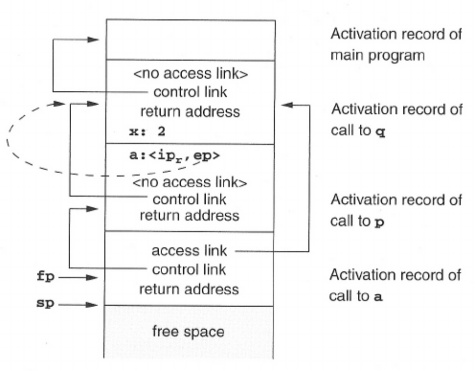
To be able to calling `r` (via `a` in the source-code), not only the address needs to be known, but also the "lexical nesting situation" of the called procedure, so that we can fill the **static link** slot in `r`'s activation record properly. At static time, that is not known, i.e., it's not known which procedure `a` represents, and that includes of course, not knowhing where the called function is placed in the scoping hierarchy,

The solution is simple: it might not be known statically, but at least is known at run-time, namely when calling `p(r)`. So when calling `p`, one not only hands over the reference to `r` as part of the parameter passing and stores that in `p`'s activation record, one also passes at run-time the "static link" for the eventual activation record of `r`/`a`. So, in a way, when calling `p`, (a pointer to) `r` together with the needed **static link** is handed over at run-time. So the RTE stores **reference** to the frame, i.e., the relevant *frame pointer*, which is here to the frame of q where r is defined. In Figure 8.23a, it's the information $\langle ip_r, ep \rangle$, where $ip_r$ represents r's instruction pointer and $ep$ refers to q's frame pointer.

This pair represents the **closure**!



(a) Closure for formal parameter a of the example



(b) After calling a (= r)

Figure 8.19: Stack-allocated closures

Now, that works well-enough. Note, however, with this design, there is now a distinction in calling sequence for calling "ordinary" procedures on the one hand, and calling procedures that had been handed over as parameters to another procedure, i.e., via closures. If one would prefer to avoid that, that may be unified in that one calls procedures in the way closures are handled hear, whether it's actually needed or not.

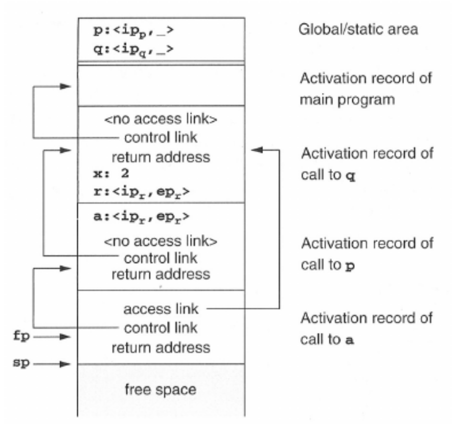


Figure 8.20: Closure: making it uniform

**Limitations of stack-based run-time environments** Procedures are one if not the **central** control-flow abstraction in programming languages. A stack-based allocation is intuitive, common, and efficient. The procedure calls and returns follow a LIFO anyway, i.e., show a stack-like behavior, and that is also supported by hardware, in the form of providing dedicated stack-pointer register and instructions that assist making the stack manipulations efficient. Indeed, a stack-based arrangement is used in many languages.

Let’s remember the purpose of the activation records. They maintain relevant information that allows to implement procedures. That involves the control-flow, remembering the return address, and the *memory needs* (local variables etc), plus more or less book-keeping information, depending on the language, to make the scoping work.

The return addresses needs a stack, that’s for sure, but that one can also put the dynamic memory need in connection with procedure calls and return on the same stack rests on one **underlying assumption:**

The data (which is part of the activation record) for a procedure **cannt outlive** the activation.

As long as that is not the case, the stack allocation of data is fine. But there can data which is needed and accessible **after** the corresponding call has returned in whose activation record the information is placed, In that case, the a stack-allocated frame would have been “removed” already when the data is needed, so it would be no longer available,

There are a number of reasons why procedure-local data can outlive the duration of the corresponding activation. Data that outlives the return of the procedure is a situation, where the data is “returned” to the outside of the procedure, explicitly or implicitly. Explicitly in the sense that there is some return statement or similar and the return value is reflected in the procedure’s type. But there are other ways as well, for instance as a side effect. Finally, if functions or procedures are returned, then, under a lexical binding regime, the body of the returned procedure may make use of variables declared in the procedure that returns the procedure. So far we dealt only with procedures as arguments of other procedures, which can still handled by a stack-arrangement (with stack-allocated closures). When *returning* procedures, that’s no longer the case. Higher-order functions, which involve both, as well as the possibility nested procedure declaration, can thereby not covered by a stack-based arrangement of activation records.

Using an explicit return on some data value does *not* per se mean the returned data outlives the activation, because it’s about the data but it’s **location**, where it’s store (here on the stack). If returning involves **copying** the data back to the callee, for instance into a variable located at the callers activation record, then all is fine. But in a language with pointers, one can retur the *address* of a local procedure variable. See the code snipped from Listing 8.16.

```
int * dangle (void) { q// return type: pointer to an int
    int x;            // local var
    return &x;        // address of x
  }
```

Listing 8.16: Dangling ref’s due to returning references

Obviously, the caller of this procedure gets hold of an address in an activation record of dangle and thus access to a part of the stack that is “no longer there”.[6] The example uses address and pointers as in C. The variable’s lifetime may be over, but the reference to the address lives on. Of course the same problem would occur in languages like Java, with referencets creating an object in the dangle procedure and then returning that (i.e., the reference to the object). I.e., it would occur, if Java would have stack-allocated objects, which of course it does not . . . . Objects live on the heap. The life-time of objects or other reference data is decoupled from the activation records of the procedures that creates them. That’s why such data is stored in a dynamic memory structure where allocation and deallocation of the memory content does **not follow a LIFO strategy**. That’s of course the **heap**!

Another thing that can break the stack-discipline of a call stack is “undisciplined” control flow, if the language supports goto. Goto’s are of course nowaday’s rather deprecated and unsupported by most languages, but C, for instance supports it.[7] Goto’s can break any scoping rules, including the procedure abstraction. Of course, also explicit memory allocation (and deallocation), pointer arithmetic etc. gives freedom of memory handling that defies a stack-discipline.

As mentioned, also *returning* functions from a call in a language with lexical scope breaks the stack discipline. In the Pascal example from Listing 8.17, a procedure defined nested in another is returned via a side-effect, storing the returned procedure in a **function variable**.

```
program Funcvar;
var pv : Procedure (x: integer);   (* procedur var     *)

   Procedure Q();
   var
      a : integer;
      Procedure P(i : integer);
      begin
         a:= a+i;    (* a def'ed outside              *)
      end;
   begin
      pv := @P;      (* ``return'' P (as side effect) *)
   end;              (* "@" dependent on dialect      *)
begin                (* here: free Pascal             *)
   Q();
   pv(1);
end.
```

Listing 8.17: Function variable (in Pascal)

The program is legal and type correct Pascal, it can be compiled and run. Doing so results in a run-time error as follows:

```
funcvar
Runtime error 216 at $0000000000400233
```

[6] Of course the part of the memory is still “there”, perhaps even the latest value of x is still there. But the memory has in the meantime reused for a different activation record with different data. At any rate, it should be considered an “illegal” access.

[7] Starting end of the 60ies, and in the seventies, there were the so-called structured programming wars. They were kicked off by one one of Dijkstra’s note, the particular one titled “Go To statement considered harmful”.

```
$0000000000400233
$0000000000400268
$00000000004001E0
```

That the programs crashes is, to some extent, a good thing, at least better than the alternative, that some random bits from a deallocated or reallocated stack area are given back without the user knowing. Basically it means, Pascal opted for a stack-based run-time environment, but supports features where that is not powerful enough.[8]

The Pascal example uses function variables and side-effects to “return” a locally defined nested function. The Go code from Listing 8.18 officially returns a locally defined function.

```
package main
import ("fmt")

var f = func () (func (int) int)  { // unit -> (int -> int)
        var x = 40                  // local variable
        var g = func (y int) int { // nested function
                return x + 1
        }
        x = x+1                     // update x
        return g                    // function as return value
}

func main() {
        var x = 0
        var h = f()
        fmt.Println(x)
        var r = h (1)
        fmt.Printf(" r = %v", r)
}
```

Listing 8.18: Function as return value

The function `g` is defined local to `f`. Under lexical scoping it uses `x`, non-local to `g` but local to `f`, and is being return from `f`. One also say, it’s a situation when `x` **escapes** it’s scope.

In languages supporting full higher-order functions, functions are treated “data” same as everything else. This is captured by say that functions are *first-class citizens.* Without higher order functions, there is a “two-class society” (or more) in the language. For instance, procedures can take all kind of language entities as a argument and can return them. Objects, references, integers, compound data, all that can be handed over and return and locally declared. But for functions or procedures, different rules apply. They may only be used as arguments but not for returns or not even that. Or, as in C-like languages, they cannot be declared in a nested way etc. At any rate, they are treated in a more restricted manner.

For higher-order functions, there are no such restrictions. Like all (other) data, function can being locally defined, are allowed as arguments and as return values for other functions. To manage the memory needs for such language, one needs what is called a **fully-dynamic** run-time environment.

[8]Pascal also supports pointer arithmethic, so also dangling references can break the stack-discipline there.

One needs **heap-allocated closures**, i.e., more flexibel ones than those we encounter earlier, which were stack-allocated, but conceptually, the closure have the same function as we descibed.

Memory management gets more challenging, In Pascal, which is not a functional language and does not support higher-order functions, it may be acceptable to run into crushing programs like the one from Listing 8.17. Using side effects on functional variables is style of program encouraged by a language like Pascal. However, returning functions and (re-)combinging them to new functions is encouraged by functional languages. Instead of trusting that the programmer will be able to handle the memory needs for that, such languages rely on an automatic memory management. Like the stack arrangement, it's part of the *run-time* environment, and thus covered in this chapter, and the automatic mechanism managing the heap is known of course as **garbage collector**.

In principle, the stack discipline can be seen as a particularly simple (and efficient) form of garbage collection: returning from a function makes it clear that the local data can be thrashed. Only the word "garbabe collection" is typically not used to refer to that part of the memory managent in the run-time environment.

**Remark 8.2.8** (NO and AO)**.**

## 8.3 Parameter passing

We discussed how the run-time environment treats procedures as a central abstraction of programming languages. Often, it results in activation records allocated on the run-time stack; sometimes that's not needed if the language is quite primitive (no recursion), sometimes allocating activation records on the stack is not possible, if the language is expressive as far as procedures are concerned (higher-order functions).

We also discussed typical designs of activation records, with the frame pointer as the "anchor" to the activation record. Besides other information, the activation records in particular contains space for the parameters of a procedure, if any.

> Parameters passing the mechanism where the caller "communicates" with the callee activation. That is bi-directional in the general case: the caller hands over the actual parameter values to the callee at call-time and, upon returning, the callee can hand back a return value.

The communication is done via appropriate slots in the activation record. Let's focus for a start on the input parameters of a called procedure, not the return value or return parameter. As we have sketched earlier, a typical arrangement is that those parameters are located "at the end" of the caller activation record resp. "at the beginning" of the callee activation record. We also discussed or sketched the concept of *calling sequence*, the steps the machine code does to realize the calls and returns, including handing over the parameters from caller to callee (and later dealing with the return value).

However, one aspect was not really discussed, namely what exactly is passed from called to callee (and back). Sure, the "parameters" are passed, but there are **different ways** to do that. There are two one basic alternatives one can do: make a **copy** of the value to hand over, or alternatively hand over from the caller to callee a **reference** to the slot where the caller keeps the value, such that the callee can access it. This latter way of using a reference is more obvious for the case the the caller passes the argument to the callee. For the return, it callee cannot just return the reference to a slot where it stores the value to be return; after all, after returning, that part of the stack is popped off and thereby not usable anymore (the reference would have to be counted as *dangling*). Still, one can also handle return in a "by reference" manner, just not in the naive way using a reference in the callee activation record. We will see later examples.

These two ways are called **call-by-value** and **call-by-reference** (and in this terminology, one speaks out calling, not returning).

**Side remark 8.3.1** (Stack-allocated closures)**.** ...

### 8.3.1 Call by-value, by-reference, by value-reference

Call-by-value is conceptually probably the simplest and clearest. Parameter passing is the act of providing the callee with a **copy** of the data used by the caller in the call. Call-by-refennce is conceptually also simple, though sometimes one finds it confused with something else, also on the internet and in text-books. In the above texts, it was formulated like that: the caller hands over to the callee a reference to the place in its activation record where the caller keeps the data being handed over. One could say shorter that in call-by-reference, a reference to the data is handed over. That's correct, of course, but it can more easily be misinterpreted.

The confusion starts if one has a programming language which supports *references* or *pointers*, as most languages do. Either explicitly and visible to the user, as for instance in C, or implicitly as in Java. In Java, instances of classes and arrays, for example, are treated as references by the language in general. A variable of a class type or of an array type does not containt the object itself or the array itself, but a reference or pointer to the data (typically on the heap).

That includes the treatment when calling function with an object or array as argument, or references in general. If we assume a language with call-by-value, if we assume a situation where a reference is handed over, is that call-by-reference? From the perspective of a compiler writer and in particular from the perspective of the calling sequence, the answer is clear: of course not! The data is *copied* from the caller to the callee, that's call-by-value without any doubt. Passing a reference *by-reference* would meean, the callee would receive a reference to the caller's slot which in turn contains a reference to the data.

The parameter passing mechanism of Java (and C, and many other languages) is **call-by-value**, period. Still, one finds statements like "be careful, in Java, objects and arrays are passed by reference, unlike data like integers or floats". And that may lead to confusion. To avoid that, the situation where references are passed by value is sometimes called **"call-by-value-reference"**, though in my opinion one would not need a special word for that: all data is passed uniformely by call-by-value, and that includes references, as well.

Does it matter? It depends, perhaps it's a bit splitting hairs, especially from the perspective of the programmer, i.e., user of a language. Passing a reference or a reference data in call-by-value language certainly feels like call-by-reference. Both for true call-by-reference and in call-by-value-reference, the callee works on the data *"shared"* with the callee, i.e., the callee's versions is **aliased**. Only in the case of call-by-reference, the data being sharing is on the stack, in the caller's activation record, in the case of a call-by-value-reference, the sharing is done via the heap. But that may be a fact internal to the run-time system and of little interest for the programmer. With the data being shared, if the calling procedure does some changes to the values of the parameters, those changes become available to the caller. This way, in a call-by-reference language, the formal parameters are commonly not just used to communicate data from caller to callee, but also to communicate information back, in that the handed over arguments have been changed. In that way, the parameters take also the task of "returning results". In a call-by-reference language, one can thus work without ever officially returning a result value (via `return v`), but works with functions of return type `void` or similar. Sometimes that's used to distinguish *procedures* from *functions*, which do return a value. Of course, one can program the same way in a call-by-value language which supports pointer or references.

In a language with call-by-value (without references), one cannot use the call-parameters for (also) communicating results back to the caller. One way of doing it is, of course, returning the value via a `return` statement. But that's not the only way. One finds also languages which support *two* kinds of parameters, for calling, as usual, and parameter(s) for returning. There are sometimes called in and out-parameters. So a procedure declaration in-parameters for receiving the arguments and out-parameters for returning the results (often multiple in-parameters but at most one out-parameter) In such a design, the caller can use call-by-value when calling. However, out-parameter is treatedin a by-reference manner. Upon calling, the caller informs the callee where it wants to find the result after the callee is finished, and for that it the callee activation record stores the address of that call-parameter, of course, the *actual* call-parameter, not the *formal* one.

Call-by-value is in a way the prototypical, most dignified way of parameter passsing nowadays, supporting the procedure abstraction. If one has references (explicit or implicit, of data on the *heap*, typically), then one has call-by-value-of-references, which, "feels" for the programmer as call-by-reference. Some people even call that call-by-reference, even if it's technically not, as mentioned earlier.

As also mentioned earlier, if the callee's activation record gets a copy of the actual parameters upon being called, one also needs a machanism to return results back, and, in according with the treatment of the procedure arguments, the result is then copied back in a pure call-by-value scheme. Also possible, however, is that a results are treated differently, see later.

Procedures or functions may operate with variables in its body, that are not handed over as parameters. Even in the simplest setup, a procedure can operate on global variables. Access to non-local variables necessitated *static* links in the activation record for languages supporting nested procedures, and to deal properly with higher-order functions, one needs heap-allocated activation records. Independent from how complex the language design wrt. procedures, there are *two* kinds of variables whose values originate from *outside* the procedure itself. One of course the input parameters which is the topic we are currently discussing. The “official” parameters of a procedure are handed over via call-by-value, call-by-reference, or some other scheme. But what about the “inofficial input parameter”, the variable that come from somewhere outside?

For the global variables, they are of course not copied, their address is globally know. For the variables originating from an surrounding procedure body, in which the procedure of the current activation record is nested in, the corresponding activation record can be located via following static links. At least that’s the situation for languages with lexical scoping. Anyway, also the values for those variables, when used in a procedure body are not copied in, i.e., even in a call-by-value parameter-passing scheme, they are treated typically by-reference.

Go, for instance, is an imperative language with call-by-value parameter passing, which supports higher-order functions and thus closures, which treats “smuggled in” variables by-reference. That is the standard treatment. If in such a language, one is unhappy with the by-reference treatment of th smuggled in variables, one can of course rewrite the procedure, add more input parameters and hand over the value officially, thereby obtaining a call-by-value treatment.

The technique to systematically promote outside variables to official parameters is known as $\lambda$-lifting. It’s mostly used in some compilers for functional languages ([2]).

### Parameter passing by-value

Let’s looks at very simple examples, using C, which is a prominent example of an imperative, procedural language using call-by-value. Listing 8.19 and 8.20 shows to versions of a procedure `inc2` taking one parameter. The function has `void` as return type, i.e. no value is returned via a return statement.

```
void inc2 (int x)
{ ++x, ++x; }
```

Listing 8.19: Integers as arguments

```
void inc2 (int* x) { /* call: inc(&y) */
  ++(*x), ++(*x); }
```

Listing 8.20: Pointers as arguments

The first `inc2` example does not work, of course, if the intention of the function is to do a double increment. Sure, the function increments its integer argument by 2, alright, but it increments a *copy* of the actual parameter, passed by value and does not do anything with the increased value otherwise. In particular, it does not return the incremented value; the procedure’s return type is `void`. The second version, what is passed is a pointer to, i.e., address of an integer value, as indicated by the parameter type `int *`. Of course, the plausible intention is not to increment that address by two, but to increment the value at that address accordingly. So the increment operation ++ is applied to `*x`, not `x`.

**Side remark 8.3.2** (Pointer arithmetic)**.** In C, it would actually be allowed to do incrementations and other calculations on addresses or pointers; that’s known as pointer arithmetic. Java and other languages would don’t offer that. Some make the distinction in terminology that Java has references whereas C has pointers. □

In C and Java (and a number of other languages), call-by-value is the only parameter passing method. Some language impose the restriction that formal parameters are **immutable**. Of course, if one passes by value a reference to an object or a piece data, the variable itself my be immutable, but it does not disallows fields of the object or the referenced data to be modified. As mentioned earlier, the passing of parameters is simply placing a **copy** of the values of the actual parameters in the slots of the activation records.

Arrays are in many languages treated as reference data. So a variable "containing" an array actually is containing a reference to a place where the slots of the arrays can be found. So, having an array-typed formal parameter means, that this reference to the array is handed over, not a copy of the array itself. So if the caller changes the content of the array, that will be visible by the caller (or any other place in the program that references that array). For instance, the code in Listing 8.21 "erases" the content of the array where the first argument references the array.[9]

```
void init(int x[], int size) {
  int i;
  for (i=0;i<size,++i) x[i]= 0
}
```

Listing 8.21: An array as argument

arrays: "by-reference" data

**Side remark 8.3.3** (Interplay of language features may muddy the water (Java))**.** The following discussion can be skipped, as it has nothing much to do with parameter passing. But it highlights that languages, for instance, Java have quite a number of features that can interplay with each other. And suddenly, something that seems clear enough, like call-by-value(-reference) seems to have unexpected outcomes.

The fact that some variables do not contain data values directly but a pointer to the place where to find the value is not visible directly to the programmer in Java. There is no need to explictly figure out the address of some place of data nor to explicitly dereference and address to obtain the value. That's all behind the scenes.

Let's try to mimic the two versions of `inc2` in Java. To not muddy the water even more with late binding of methods, let's use static methods (see Listing 8.22).

There are two variants of `inc2`, one with its paramter of type `int` and one of type `Integer`. The latter expects an instance of the class `Integer` as argument, i.e., a reference to such an instance.

```
public class Inctwo {
    public static void inc2 (int x)     {++x;++x;}
    public static void inc2 (Integer x) {x++;x++;}
    public static void main(String[] arg) {
        int     x1 = 0;
        Integer x2 = new Integer(0); // deprecated
        inc2(x1);
```

[9]At least it erases the array, if the second parameter is actually the array size. Otherwise either not all of the array is set to zero, or an out-of-bounds situation occurs, which is something to be avoided, and checked for . . .

```
        inc2(x2);
        System.out.print(x2);          // guess what's printed
    }
};
```

Listing 8.22: Call-by-value or by-value-reference, or what?

There are some aspects of the code unrelated to the issue at hand, namely parameter passing. One is that there are two methods called `inc2`. Depending on whether the method is called with `x1` as argument or `x2`, the appropriate one is chosen. The parameter for both versions is of different type, `int` vs. `Integer`, and that's good enough for disambiguation. That's an example of *overloading*, more precisely, of **method overloading**, a variant of **polymorphism**. We brushed upon overloading in the chapter about types and type checking. Another aspect not crucial for parameter passing is the fact that the methods are **static**, the same would occur when using late-bound methods.

What's then the issue? According to the discussions about call-by-value used on reference data, one could suspect, that the value of `x2` printed at the end is `2`, i.e., the second version of the method `inc2` in the example corresponds to the second version of the C-code, passing a refence by value to a called procedure or methods. Call-by-value-reference is also what happens there. However, the printed value is not `2`, but `0`. So the method behaves as if it where call-by-value on an integer value, not as the counter-part in C.

The reason(s) for that are actually quite simple, and they have also not much to do much with parameter passing. You may try to reflect on why the result is `0` before reading on.

The reasons have to do with with the nature of the `++` operation and some conversions that are done behind the scene.

First to the `++` operator. It's *not* defined on integers, i.e, expressions like `5++` are illegal. What is allowed are `++x` and `x++`, the pre-increment resp. post-increment of the integer content of the variable `x` (the difference between pre- and post-increment are not so relevant in the context of this discussion). If `x` is of type `int`, the operator directly takes increments the content of the variable by 1 and stores the result back to `x`. So far, so obvious.

The operator, however works also on variables type by `Integer`. An expression like `(new Integer(5))++` is illegal; as said, `++` works on variables only. In particular `++` is *not* interpreted or translated to invoke a "method" on the integer object, perhaps like the following:

```
Integer x = new Integer(5);
int h     = h.intValue();      // that's possible
x.setIntValue(h+1);            // that's impossible
```

Listing 8.23: Pseudo-Java

That's illegal in Java. Instances of `Integer` are *immutable*, in particular, they don't have a set-method; they do have a get-method, called `intValue`.

If, however, `++` is applied to a variable of type `Integer`, what then happens is that the object of type `Integer` is **converted** to the corresponding integer value of type `int`, and in that way `x++` in the second method is well-typed and works, with some conversion going on behind the scenes. This implicit conversion can be interpreted in leading to a form of *polymorphism*. The line between overloading and conversions of that kind is a bit blurred; both count among so-called *ad-hoc* polymorphism. We shortly discussed that in the typing chapter.

That should make clear what happens in Listing 8.22. The reference to the integer object is passed by-value, the body operators on the variable `x`, the formal parameter of type `Integer`, which contains a copy of a reference, at least at the beginning. Operating on `x` doing `x++` does not change the state of the integer object, but creates a new one, to which the parameter `x` points, thereby severing the connection to the caller's reference kept in `x2`.

In general, the remark still holds: in a call-by-value language, passing references as values makes it behave like call-by-reference, though it technically is not. If, for instance, passes a "real object" (not a special case of an immutable value object as here with some specific conversions going on) an the callee mutates instance variables in that object, then of course the calleer will see those changes. But for the special case if `Integer` objects, the code of C with pointer behaves different from the "analogous" code in Java with references. In connection with that: as said, integer object are immutable, and for **immutable** data call-by-reference and call-by-value are the same anyway. □

### Call-by-reference

The main alternative to call-by-value in procedural, imperative languages is **call-by-reference.** Instead of handing over (a copy of the) value, the callee receives a reference to the actual parameter. That may be advantageous especially for large data structures. As disucssed earlier, call-by-reference "feels" much like doing call-by-value with a reference (as done for instance in Listing 8.20), but it's not the same. It's conventional for call-by-reference, that the **actual parameters have to be variables**. Calling a procedure by reference on, say 5 makes not much sense; what's the address of 5 anyway. . . [10]

```
void P(p1,p2) {
  ..
  p1 = 3
}
var a,b,c;
P(a,c)
```

Listing 8.24: Call-by-reference

A corresponding layput of an activation record is shown in Figure 8.21.

### Call-by-value-result

As said, call-by-value and call-by-result are the two main alternative for classic procedural, imperative languages. But there exist also lesser known alternatives. One is **call-by-value-result**. As said at the beginning, the communication between caller and callee is a bi-directional thing, passing arguments from the caller to the callee and returning results back. When calling the actual parameter are handed over to the formal parameters, when

[10]Fortran actually allows things like `P(5,b)` and `P(a+b,c)`.

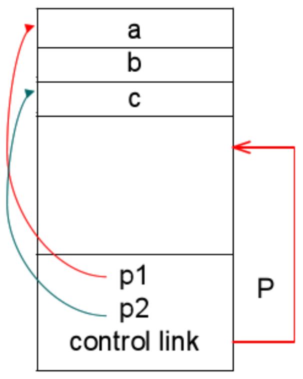


Figure 8.21: Call-by-reference, activation record layout

returning the content of the formal parmeters is hand back to the caller. Some languages, like Ada, are quite explicit about that fact, in that it distinguishes between `in` and `out` parameters.

The strategy here is less explicit about that, it just has one kind of parameters, but they are used in a two-way communcation. The mechanism is also known as **copy-in-copy-out** or **copy-restore**.

When calling, the arguments are copied in, i.e., that part works as in call-by-value. However, to get potential results back, the mechanism works like call-by-reference, as it uses the reference of the actual parameters to place the returned values. Let's remember the example from Listing 8.19. Under call-by-value, the procedure changed the copy of the actual argument, but if the intention was to get a double increment as a side-effect, that did not happen. Under call-by-value-result, that's exactly what would happen, so it would behave more like call-by-reference. But of course, similar to call-by reference, it makes no sense to call the function `inc2` under this regime like `inc2(4)`. To achieve the intended effect, one would have to do `x=4;inc(x)`. The `x` here is of course a different `x` than `inc2`'s formal parameter from Listing 8.19, the usual scoping rules still apply.

That may sound straightforward enough. But there issues and corner cases where it is suddenly not so clear anymore. *when* are the value of actual variables determined when doing "actual $\leftarrow$ formal parameters": when calling? .when returning

For instance, what would or should happen in the code of Listing 8.25? The example has some form of *aliasing* in its arguments, using `a` for both argument. In general, call-by-value result gives the same results as call-by-reference, unless *aliasing* "messes it up" as in Listing 8.25[11].

```
void p(int x, int y)
{
  ++x;
  ++y;++y;
}
```

[11]One can make the argument though, that call-by-reference would be messed-up in an aliasing situation as well. Though at least the answer in a function as in Listing 8.25 under the call-by-reference analogon would be clear, it would be 4.

```
main ()
{  int a = 1;
   p(a,a);    // :-O
   return 0;
}
```

Listing 8.25: Call-by-value-result example

### Call-by-name

The last parameter-passing mechanism covered here is **call-by-name.** We present it in a C-like syntax for a C-like language. When calling a function with an argument, it's not the value that is handed over, but the name of the argument, like the name of the argument. So when calling `f(x)`, with `x` containing 4, the formal parameter of `f` is replaced by `x`, not by 4 or a reference to the variable $x$. Calling a function therefore works with a form of *substitution.* It resembles also macro expansion, but scoping still applies. In that scheme, the value of the actual parameter is *not* fetched or calculated *before* actually used. In the other hand: if the argument is needed more than one it needs to be *recalculated* over and over again. It's therefore called **delayed evaluation**.

A possible way to implement that scheme is the following I case the actual parameter is a compound expression, it's representend as a small procedure (also called *thunk* or *suspension*). That then will be "called", i.e., the expression evaluated, when the function needs the value of the parameter. If the actual parameter is a variable, not a compound expression, one can optimize that and pass the variable directly, thus avoiding turning it into a thunk. Indeed, with a variable as actual parameter, call-by-name results in the same behavior as call-by-reference.

In a few examples below, call-by-name will lead to confusing behavior and programs hard to understand. In principle, call-by-name seems innocent enough. Textual replacement (or substitution) of the formal parameter variable by the variable that represents the actual parameter looks straightforward. For expression arguments, replacing the formal parameters in the body by the argument expression seem likewise easy. The only price to pay in that case is that at run-time, the expression may have to be evaluated more than once. Indeed, there is an optimization of call-by-name that avoids that. In case the expression needs to be evaluated, the result is remembered ("memoization") and fetched when needed gain.

All that is clear enough, but it becomes rather less so, if evaluation of the argument **has side-effect.** Indeed, remembering the result of the evaluation for later reuse as in call-by-need makes only sense, if the later evaluations will result in the value and if the expression itself it has no side effects. If an expression have side-effects, of course, executing it twice is different from executing it once (or not at all if not needed). "Expressions" with side-effects, like `(x++) + (15*y)`, are perfectly fine in many languages, like C or Java (which of course have call-by-value).

What some languages do not supported is passing procedures as arguments, which of course also may have side-effects. Of course in the above example, one could view the compound expression as a call to (built-in) procedure "+" which takes to arguments. The first one is the application of an increment-operation on the argument `x` and the second one some side-effect free calulation.

What makes the examples below a bit hard to decypher is exactly that: **side-effects** in combination with handing over an argument that requires calculation, a calculation, that under call-by-name, is delayed and potentially done multiple time. The argument that requires calculation in the examples will be an array access `a[i]`. Even if in "C-like" languages, one cannot hand over procedures, arrays can be seen as procedures. An integer-indexed integer array is a realization of a function from of type `int`→`int` (with a finite domain and mutable).

The example from 8.27 shows a function with one integer argument. Additionally, the changes the argument, incrementing it:

```
void p(int x) {...;  ++x; }
```

Listing 8.26: An array as argument

The example then calls the procedure with `a[i]` as argument. Under call-by-name, not the value corresponding to the array access is handed over, but the evaluation is *delayed*. Another way of seeing it is that `++x` in the body of the procedure is executed `++(a[i])`.

```
int i;
int a[10];
void p(int x) {
  ++i;
  ++x;
}

main () {
  i = 1;
  a[1] = 1;
  a[2] = 2;
  p(a[i]);
  return 0;
}
```

Listing 8.27: Call-by-name example

In combination with the side effect, including a side effect that changes `i`, that can be pretty confusing. That's not the only side effect the procedure does, it additionally incrementing `i` in Listing 8.27, So the delay in the evalutation of `a[i]` is not just a delay. Since the "argument" of the array access it mutated, it also accesses a slot that reflects the content of $i$ at the time of the access.

The next example from Listing 8.28 is basically analogous, and perhaps slightly a little less artificial. It uses a procedure called `swap`, whose body executes a typical way swapping the content of two variables, here containing integers and here corresponding to the formal parameters `a` and `b`. That can be achieved by introducing an auxiliary variable, here called $i$ and doing the obvious three swapping steps:

```
  i=a; a=b; b=1;
```

Under call-by-value, the procedure would not work, for the same reasing that `inc2` from Listing 8.19 did not work. How about call-by-name?

```
int i; int a[i];

swap (int a, b) {
  int i;
  i = a;
  a = b;
  b = i;
}

i = 3;
a[3] = 6;

swap (i,a[i]);
```

Listing 8.28: Swapping

However, applying the swap-procedure under call-by-name may run into problems. For instance, applying it to the the pair `i,a[i]` does not swaps the contents of `i` and the slot `a[i]` of the array.

Note that the example uses a local and global variable `i`, but that's not really the problem, at least not if we are clear about how call-by-name is supposed to work. Earlier we said, parameter-passing can be understood as replacing the variables "textually" by the arguments. For that replacement, though, some fine-print applies. If we simply replaced or substituted the first formal parameter `a` by `i`, then the first assignment `i=a` would be `i=i`. That's **not** how one should interpret "use the name of the actual parameter" or "textual replacement". At least not under **lexical scoping**. Under lexical scoping[12], the global and the local `i` are different variables, and they reside also in different parts of the memory. Indeed, the choice of names should not matter, it's the run-time system task to keep them appart and track which one is meant at each point. For instance, the auxiliary local variable `i` might as well have been called `aux` or `h` or other name that is not in use otherwise. One way if understand substitution as explanation call-by-name is that the substitution must avoid variables being re-bound. If we simply interpret the program as writing `i` for `a` (similarly for the other function argument), the variable `i` representing the argument would suddenly, looking at the body after the replacement, be **rebound**. One also says, by substituting in this careless manner, the variable would be **captured**, namely captured by the local declaration `int i` (which would not happen of the swap-procedure would have called the auxiliary variable `aux`. This capturing corresponds to dynamic binding, and we are doing call-by-name with static binding. The form of substitution adequate for lexical binding, that avoid that problem is known as **capture-avoiding substitution**.

So, we silently assume capture-avoiding substitution or assume that the swap uses `aux` instead of `i` from the start. At any rate, "capturing" `i` is *not* what prevents `swap` to properly swap the content of its arguments in Listing 8.28[13]

Now, avoiding to confuse the global `i` with the local auxiliary variable, and carefully doing the steps

[12] actually also under dynamic scoping
[13] Of course, if $i$ would be recaptured, that would additionally be problematic. But that confusion would not be caused by the combination of delayed execution and side-effect, it would be caused by coincidentally using the variable name `i` twice.

```
 i_aux = i;
     i = a[i];
  a[i] = j
```

Listing 8.29: An array as argument

should make clear what the result of the procedure is, and it's not a swap.

**Side remark 8.3.4** (Capture-avioding subsitution)**.** All different variables, implementation

With substitution as parameter passing, one has also to watch out that not at each place where a variable is syntactically allowed, the formal parameter, also some more complex expressions are allowed. We have seen a similar restriction for call-by-reference. Listing 8.30 shows some Pascal program making use of call-by-name and Table 8.2 shows, what's allowed and what not. The situations should be fairly self-explanatory.

```
procedure P(par): name par, int par
begin
  int x,y;
  ...
  par := x + y; (* alternative: x:= par + y *)

end;

P(v);
P(r.v);
P(5);
P(u+v)
```

Listing 8.30: Call-by-name

| | v | r.v | 5 | u+v |
|---|---|---|---|---|
| par := x+y | ok | ok | error | error |
| x := par +y | ok | ok | ok | ok |

Table 8.2: Not everything makes sense under call-by-name

### Lazy evaluation

The previous examples and discussion may give the impression that call-my-name is more than a bit weird and perhaps historically interesting, used in Algol 60 or similar[14], but otherwise the evolution of programming languages left behind call-by-name as a bad idead, forgot about it, and the only place where it occasionally shows up is in academic compiler construction or programming language course, when discussing it as a parameter passing curiosity.

As far as imperative languages is concerned, there is a point. We have seen that combining side effects and delayed evaluation leads to behavior hard to understand, and no obvious

[14]It has even been called a **misfeature** of Algol 60.

upside. If the procedure depends on concrete arguments that can change, it matters when the procedure is called, before or after a possible change to its arguments. And if the procedure has side effects itself, it matter how often it is called. Of couse the latter point is quite common, for instance a procedure doing a double increment like the call-by-reference version `inc2` from Listing 8.20 is intented to be called multiple times, and if called 5 times on the same argument, it has incremented that by 10. Of course, when handing over a "functional impression"[15] like `a[i]` in the examples, how many times that expression is evaluated is far less obvious, and likewise, what value `i` will have when being evaluation. In other words, that can get much more confusing than invocing `inc2` a fixed number of times (or in a loop, or at the beginning of a procedure as a form of counter).

However, **without side-effects** the problems disappear. Still it may be unclear how often a functional argument is evaluated (if at all), but it does not matter. In a purely functional setting, invoking evaluating a function application now or later will give the same result. Indeed, no matter how many times it will get evaluated, it always gives the same result. Call-by-value corresponds to a scheme, that arguments are evaluated at the time when they are passed over to a procedure as parameters. Call-by-name hands over the arguments "as is", potentially unevaluated, and delays their evaluation only when used. But it does not matter when and how often the arguments are evaluated. It does not matter wrt. the returned value, that is. Of course, if the unevaluated argument is a complex computation, evaluating it repeatedly is not the best of idea, efficiency-wise, given the fact that the result is always the same. Better would be to remember the result after the first time it's needed, for for all potential subsequent uses, just look it up.

That optimization of call-by-name, in a functional setting, using memoization, is called **lazy evaluation** or **call-by-need**

The principles has most prominently be realized in Haskell, a purely functional language with lazy evaluations. Haskell has other interesting features as well, but sticking to parameter passing, let's at least answer the obvious question: what is lazy evaluation good for? If it basically does not matter if things are evaluated lazily (call-by-need) of eagerly (call-by-value), why bother?

Well, the answer comes from that fact that it lazy and eager evaluation does **not** lead to the same behavior. Thought it's true that given two functional programs, one evaluated lazily, the other eager, they give the same result when returning. But doesn't the latter just contradict the previous sentence that both behave differently? Actually it's not a contradiction: both evaluation strategies do give the same result **if terminating**. However, there are situations, when lazy evaluation **terminates** but eager evaluation does not (not the other way around).

That sounds like an awfully miniscule difference, and who needs non-terminating (functional) programs anyway, at least non-terminating when evaluated the wrong way?

Actually, there is an area where that difference plays a crucial role. It allows ot program with operating on **infinite data structures.** The simplest example for that are infinite versions of lists, known a *streams*. Perhaps it's better to say, streams are potentially

[15] It's of course not a function but an array access expression, but it behaves as applying a function `a` to an argument `i`.

infinite lists, not *actually* infinite lists, list, where there is always a next element or tail, when needed. I.e., the data structure itself is “lazy”. Of course, infinite lists cannot be passed or operated on in a *by-value* manner. After all, the infinite list in evaluated form corresponds to an infinite amount of data. and calculating all the data elements **would not terminate.**[16] But under a lazy interpretation, one only evaluates or operates on one element of a stream after each one when actually needed.

Without exploring that further, we simply show an example of a stream of integers, actually, the infinite *fibonacci series*.

```
magic :: Int -> Int -> [Int]
magic 0 _ = []
magic m n = m : (magic n (m+n))

getIt :: [Int] -> Int -> Int
getIt []     _ = undefined
getIt (x:xs) 1 = x
getIt (x:xs) n = getIt xs (n-1)
```

Listing 8.31: Lazy evaluation/streams

## 8.4 Virtual methods in OO

In the following we shed some light on aspects of the run-time system relevant for object-oriented languages. Not too much light, though, and only for a few aspects of object orientation. It’s basically for one aspect for some main-stream object-oriented languages, like C++ or Java. Those are class-based languages with class inheritance and the aspect we look at in this context is late binding or dynamic binding of methods. We use the terminology of **virtual methods** here, which is common for C++ and is also used for languages like Object Pascal. However, the terminology “virtual” as well as the concept is older. It originates from *Simula*, the first object-oriented language, developed in Oslo by Ole-Johan Dahl, Kristen Nygaard, and colleagues [1], who got the Turing award for the contribution.

For Java, one often does not use that word, but conceptually, methods in Java are late-bound by default, i.e., in they are *virtual* in C++ terminology. We use the word virtual method, late bound method and dynamically bound method interchangably.

We earlier also mentioned *dynamic dispatch*. Dispatch is what the run-time system has to do when calling a procedure, function, method etc., i.e., *jumping* to the beinning of the code of body of the procedure (plus performing the steps of the call sequence). Determining the jump target, i.e., locating the code of the procedure, can be done at compile time or a run-time. That’s **static dispatch** resp. **dynamic dispatch**. Preferably, efficiency-wise, is static dispatch. If the compiler can determine the jump-target, then it can produce code with the jump address “hard-coded” into the jump-command. When writing, for intstance, `x.m()`, establishing the connection between the source-code level name `m` of the method and the corresponding method body, ultimately the address in the code section, that’s also called *binding*. Binding names to addresses is, of course, more general than for methods or

[16] There are simply ways to simulate lazy evaluation in an eager language. It’s analogous to what we called earlier thunk or suspension.

procedure names only. The association of `x` with its address is likewise called `binding`. We have mostly looked a statically bound variables (also called lexically bound). For instance, the use of static links to locate the proper address of a statically bound variable in languages with nested procedures.

Static binding is impossible for late-bound methods, resp. late-bound just means the binding is done at run-time, i.e., dynamically not statically. The concept of late bound method is central for the concept of [... continue here... ] The need for dynamic binding for virtual methods is ultimately can be explained by the

```
class A {
  int x,y
    void f(s,t) { ... F_A ... };
  virtual void g(p,q) { ... G_A ... };
};


class B extends A {
  int z
    void f(s,t) { ... F_B ... };
    redef void g(p,q) {... G_B ...};
    virtual void h(r) { ... H_B ...}
};


class C extends B {
  int u;
  redef void h(r) { ... H_C ... };
}
```

Listing 8.32: Virtual and non-virtual methods + fields

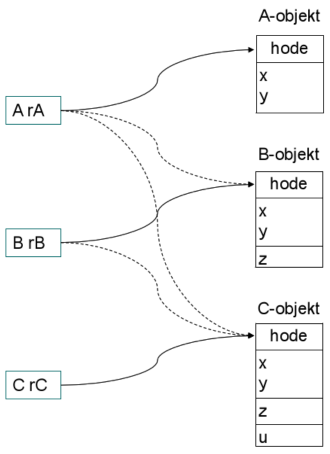


Figure 8.22: Layout

The code sketches a situation with inheritance, more precisely class inheritance. It's not exactly Java or C++ code, for instance the keyword `redef` is used here to highlight a situation which is more commonly called method **overriding**. Simula, though, used the terminology and keyword `redef`, though that did not stick.

The code shows virtual methods and static ones (the latter called `f`). The "boxes" on the left of the picture, illustrate variables called `rA`, `rB`, and `rC`, typed with `A`, `B`, and `C`, respectively. The identifiers A, B, and C are, in languages like Java etc, at the same time the names of classes that are used to created instances or objects. In the material about typing, we discussed that in many (statically typed) class-based object-oriented languages, the class names not only serve the role to denote the class, but also play the role of being (the name of) a static type.

The instances on the right are shown containing their instance variables. That corresponds also to the way objects stored in memory, typically allocated on the heap.

The corresponding methods, which are, with the field, sometimes also called "members" of an object, are *not* shown in the picture as being part of the instance. Conceptually, that's not wrong. Sometimes objects are explained as a construct "*containing*" data together with the code that operates on the data, like a bundle of fields and methods. This mental picture is fine, though we know already, that objects (with the content of the fields) are allocated on the heap, whereas the code of the methods resides in the (static) code area.

Of course, an object "containing" a method can be more realistically interpreted also in that the heap object contains, besides the content of the fields, also addresses of the methods it offers. This is a possible design of the run-time environment for objects, and is known as *embedding* (the methods). This obviously allows late-binding and dynamic dispatch: at run-time, access the object, find the slot containing the intended method implementation, and then dispatch to there.

Embedding of methods is, however, not the way languages like Java or C++ solve the late-binding issue, and this short section is mostly about non-embedding alternatives. In some languages or situations, however, there is no alternative to embedding of methods. That's if method-update is supported. Method update is different from method overriding in inheritance-situations. Fields (unless immutable) can be updated, i.e., their content can be replaced by a new value at run-time. In Java and C++ etc. which methods are supported (including which code is executed when such a method), that's fixed when instantiating a class. It's not possible to "replace" the code of a supported method at run-time (nor is possible to add a new method or remove one at run-time). In languages that support that, one needs to embbed the methods into the object (in the mentioned sense of embedding a pointer to the intended code).

One way of describing the situation in Java (which is a statially typed language) is that when.an instance of a class is instantiated, that fixes its *run-time type* (seeing the class again as a type). Now, if the run-time system knows the class of which aan object is an instance of, its run-time type so to say, then that can be used to perform a dynamic dispatch. Of course, to do so, the run-time system need to keep information about how the classes relate to each other and when a method overrides another. In a language with single (class) inheritance, that means, the run-time envorinment has a representation of the **tree of inheritance** available plus an overview over the override information. Then the classes contain pointers to the code of the methods they implement. Then the dynamic dispatch could be done by navigating the tree: if a method is called "on" and object, the corresponding run-time type in form of the class is consulted, if the class implements the methods, it will contain the pointer to the code, which is then used for the dispatch. If not, the parent class is consulted; each class, except the top-level class (Object in Java) has a unique parent class in a single-inheriance language. In this way, the look-up will eventually able to find the corresponding code.

In a statically typed language (and if type-safe), it is guaranteed that the seach will find the code. It's just not statically known in which class it belongs to, that's why the run-time system searches the tree at run-time.

That leads to the following reprentation: each object keeps a link to its class, each class keeps an link to all the method it itself implements plus a link to its parent, if any (plus pointers to sub-classes).

That's a plausible solution, though it can be improved. In particular improving on the "search-the-tree" part. Now that it's mentioned, the improvement is also pretty obvious: at dispatch-time, **don't** send the run-time **searching** for the code in the inheritance tree hierarchy. If a method is not directly supported by a class, but inherited from the super-class, or super-super-class etc., just find out at compile time already from where it comes and copy in the corresponding address into the class. That's basically it.

Thus the object points to a data structure which contains pointers to methods, not more, It's not therefore not a pointer to the "class" of the object, but to the relevant information needed to do the dispatch. This table-like structure is also called **virtual function table**. This is a standard design in standard object-oriented language (= class-based, single-inheritance).

Back to the "example" from Listing 8.32 and Figure 8.22.

The following tables summarize and repeats information of the previous picture. $r_A$, $r_B$ and $r_C$ are meant as variables of *static* type $A$, $B$, or $C$. It lists which code can be executed in each case. For the late-bound methods, that static type does not provide information to be sure which code is meant. For instance, in the second table, calling $h$ in a variable with static type $B$ (there written $r_B.h$) can mean to execute $H_B$ or $H_C$. It is, because being of a variable of type $B$ can also contain objects of run-time type $C$. Another way of seeing the same thing is: an object of static type $C$ is *also* of type $B$, and of type $A$ (in the given example). In Java, it additionally would be of type `Object` as the supertype of all class types. That's a sitation in a statically type language were a well-typed language construct (a variable, an object) has more than one type ($A$, $B$, $C$, and maybe `Object`, where we use the class names in they role as types. So, this is another from of polymorphism (we touched upon overloading and coercion als other forms of polymorphism before). This one is known as *subtyping* polymorphism or *inclusion* polymorphism. It is typicall for object-oriented languages, in particular in the form here, where inheritency (which connects classes and is about code reuse) is connected to *subtype polymorphism* in that inheritance between classes implies subtyping between the corresponingly named types. So the names are criterion to decide when a type is a subtype of another, namyl the class corresponding to the subtype is in a inheritance relation to the class corresponding to the supertype. So that is known as *nominal* subtyping. We have seen the distinction between name-based (i.e., nominal) and structural criteria to compare types when discussing when two types are "equal".

### Virtual function table

- static check ("type check") of $r_X.f()$
  - for virtual methods: `f` must be defined in $X$ or one of its superclasses

| call | target |
|---|---|
| $r_A.f$ | $F_A$ |
| $r_B.f$ | $F_B$ |
| $r_C.f$ | $F_B$ |

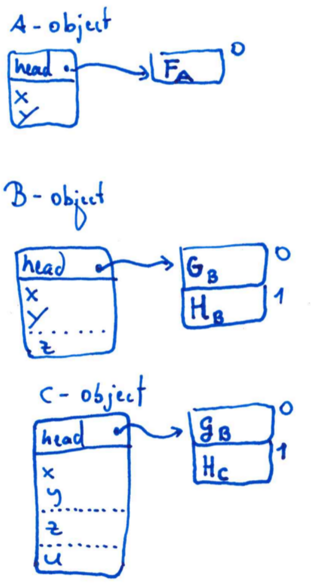
| call | target |
|---|---|
| $r_A.g$ | $G_A$ or $G_B$ |
| $r_B.g$ | $G_B$ |
| $r_C.g$ | $G_B$ |
| | |
| $r_A.h$ | illegal |
| $r_B.h$ | $H_B$ or $H_C$ |
| $r_C.h$ | $H_C$ |

Table 8.3: Call targets for non-virtual method $f$ and virtual methods $g$ and $h$

- non-virtual binding: finalized by the compiler (static binding)
- virtual methods: enumerated (with offset) from the first class with a virtual method, redefinitions get the same "number"
- object "headers": point to the class's **virtual function table**
- $r_A.g()$:

```
call r_A.virttab[g_offset]
```

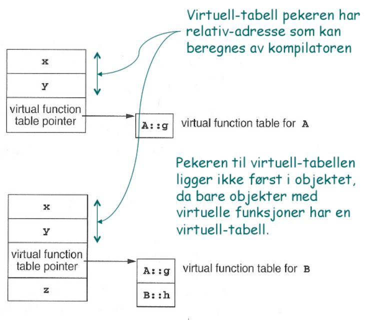
- compiler knows
  - `g_offset = 0`
  - `h_offset = 1`



## Virtual method implementation in C++ B_frame

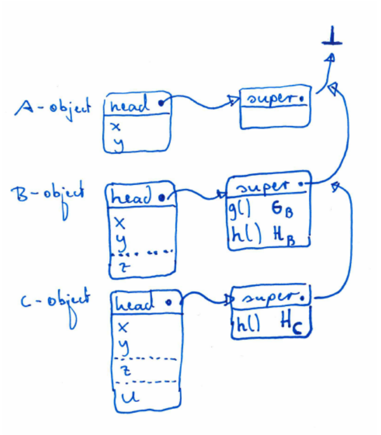
- according to [3]

```
class A {
  public:
  double x,y;
  void f();
  virtual void g();
};

class B: public A {
  public:
  double z;
  void f();
  virtual void h();
};
```


Virtuell-tabell pekeren har relativ-adresse som kan beregnes av kompilatoren
x
y
virtual function table pointer
A::g
virtual function table for A
Pekeren til virtuell-tabellen ligger ikke først i objektet, da bare objekter med virtuelle funksjoner har en virtuell-tabell.
x
y
virtual function table pointer
z
A::g
B::h
virtual function table for B

**Untyped references to objects (e.g. Smalltalk)** **B_frame**

- all methods *virtual*
- *problem* of virtual-tables now: virtual tables need to contain all methods of all classes
- additional complication: *method extension*, extension methods
- Thus: implementation of `r.g()` (assume: `f` omitted)
  - go to the object's class
  - *search* for `g` following the superclass hierarchy.



## 8.5 Garbage collection

The dynamic part of the memory consists of the stack and the heap (remember the general layout shown in Figure 8.1. The stack we discussed at some length, but remark also which kind of data are allocated on the heap, basically dynamic data whose livetime is not aligned with the activations of procedures.

**Dynamic** memory means that it's allocated a deallocated at *run-time*, resp. portions of it. There are different alternatives for that, including that it's left to the programmer, i.e., the allocation is done manually with specific commands ("alloc", "free"). That seen as error prone. **Garbage collection** is a part of the run-time environment that manages the dynamic memory automatically, in particular reclaims unused parts, it collects the "garbage".

The run-time stack of course is an automatic and important management system for dynamic memory. Popping an activation record frees and reclaims the corresponding portion for reuse. One could call it therefore some form of garbage collector, but that's not normally not called garbage collection.

Instead the garbage collector takes care of the heap. As the allocation and deallocation of data on the heap is less disciplined, not following a LIFO discipline, it's more complex and less efficient.

part of the *dynamic* memory

contains typically: objects, records (which are dynamocally allocated), often: arrays as well, for "expressive" languages: heap-allocated activation records: coroutines (e.g. Simula), higher-order functions

it should be noted"heap" unrelated to the well-known heap-data structure from A&D

To do automatic garbage collection, the run-time environment need to figure out what is garbage and what is not. That is basically impossible in languages with pointer arithmetic, like for instance in C. We have not really discussed pointer arithmetic, we have discussed the phenomenon of *dangling references* in Listing 8.16, which is a problem of the stack, not the heap. For the compiler, pointers and pointer arithmetics are not problematic. If the programmer can access and calculate directly with addresses, that's at a very low level of abstraction, and so the compiler has not much to do there. And since operating on the heap is given in the hands of the programmer, it's the programmer's task to use, reusing, and free the memory the way it suits the needs of the programmar. Of course, there is support by the compiler by compiling commands like `alloc`, but a garbage collector cannot reasonably figure out which memory locations are free and for not.

We have seen also that returning procedures or higher-order functions lead to the need of heap-allocated memory (remember for instance the code from Listing 8.17).

It's not unreasoable, in a setting with pointers and pointer arithmetic, to expect from the programmer to clean up unused memory. After all, one hard to have it both: low-level access and free manipulation of memory *and* support of the compiler to protect oneself to misuse the freedom.

For higher-order functions, the situation is different. Returning functions and especially higher-order functions are a powerful abstraction. It leads can lead to complex access patterns on the heap and in the address space. They would basically be unusable of one offers the user that abstractions, but at the same time expecting that the user cleans up the activation records, which would require low-level understanding of what's going over and would make higher-order functions error-prone and probably unattractive.

In other words, garbage-collection is a must-have in such languages, and indeed, garbage collection is pretty old, already dating back to 1958 and Lisp.

### Some basic design decisions

We don't go into much details concerning garbage collection, but let's start some basic design decision. As part of the run-time system, garbage collection is done at run-time. Still, it works **approxiative,** that's a (general) feature it shares with *static* analysis, even if it's dynamic. The approximation realizes the non-negotiable condition to assume *memory safety*

**never** reclaim cells which **may** be used in the future.

.

The reason why that garbage collection works approximatively is of course, that it cannot foresee the future. The compiler cannot know if a particular address will be accessed in the future or not, as it does not know which steps will be done.

There are different ways to do garbage collection, one basic decision is whether it involves **moving** objects during the clean-up or not. By objects we mean, not just objects in the OO sense, but all kinds of data times allocated on the heap.

If one never moves objects, that will lead to **fragmentation** of the heap space. One may end up with many stretches of free space, but so small that they are often not usable for larger objects, and one ends up with all heap space effectively gone (or should one say, non-effectively . . . ), even if the space is still sprinkled with many small pieces of free space.

That can be avoided if the garbage collected moves objects which are still needed. There are different ways to do that, we look at two variations of that, known as **mark-and-sweep** and **stop-and-copy**. That requires not only the extra administration/information needed all reference of moved objects need adaptation all free spaces collected adjacently (defragmentation)

Another design decision is to **when** to do garbabe collection. The most obvious answer here would be: "do it when it's needed". Alternatives may be to somehow "continuously" do it, always cleaning up "a bit".

Also there are different ways **how** the garbage collector gets or mainains information about definitely unused resp. potentially used obects? It could 'monitor" the running program, resp. the interaction of the program and the heap, to keep up-to-date all the time. Alternatively, the garbage collector could inspect (at approriate points in time) the *state* of the heap

Both covered styles of garbable collection are based on the following observation: heap addresses are only **reachable** in the following two ways: either **directly** through variables (with references), kept in the run-time stack (or registers), or **indirectly** following fields in reachable objects, which point to further objects . . .

In this way, one can see a heap heap as a **graph** of objects. The entry points to the graph, i.e., data *directly accessible* from the program are are called the roots and all the entry points are the **root set**.

### Mark and sweep

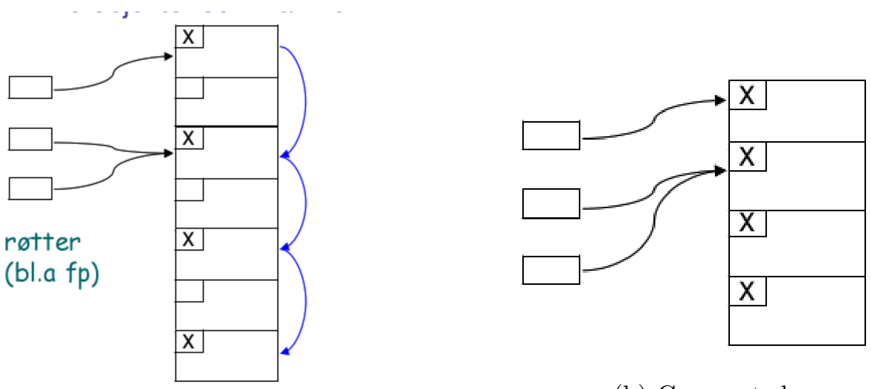
Mark and sweep is pretty simple and works in two phases, the marking and the sweeping phase, of course. The marking phase is basically doing graph search, starting from the the root set: find reachable objects, **mark** them as (potentially) used. One bit of information is good enough for marking. The search can typically be done as depth-first search of the graph.

The layout (or “type”) of the objects need to be known to determine where pointers are, and to follow them. One should keep in mind that doing a DFS requires a *stack*, in the worst case of comparable size as the heap itself . . . .

After marking one nows all the potentially used objects. Implicitly one has determined also all definitely unused ones, i.e., the garbage, namely all unmarked one. Now, having determined the garbage as the pool of unmarked objects, however is not in intself useful, as one has the information not readily at hand.

e That’s when the second phase comes in, the **sweep**. The garbage collected gos through theh eap again, this time sequentially, i.e., no more complicated graph search is needed. It collects all unmarked objects in the so called **free list**, while all objects remain at their place. If the run-time environment needs to allocate a to allocate new object it grabs a slot from free list.

Afterwards, one can add as optional third phase **compaction** to avoid fragmentation. That **moves** all the non-garbage to one place and the rest is one big and unfragmented free space. Moving the object obviously involves to adjust pointers.



(a) Marked

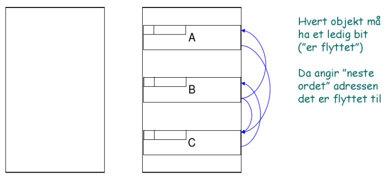
(b) Compacted

Figure 8.23: Mark (and sweep) and compact

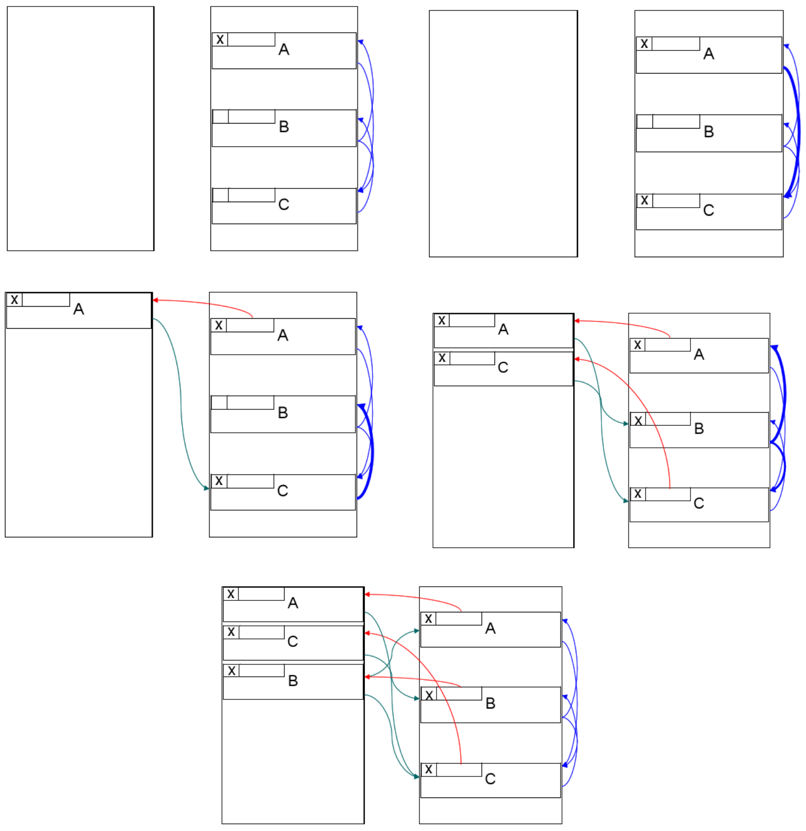
## Stop-and-copy

**B_frame:**

- variation of the previous compaction
- mark & compaction can be done in recursive pass
- space for heap-managment
  - split into *two halves*
  - only one half used at any given point in time
  - compaction by copying all non-garbage (marked) to the currently unused half


A
B
C
Hvert objekt må ha et ledig bit ("er flyttet")
Da angir "neste ordet" adressen det er flyttet til

## Step by step



# Bibliography

[1] Dahl, O.-J., Myhrhaug, B., and Nygaard, K. (1968). Simula 67, common base language. Technical Report S-2, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway.

[2] Johnsson, T. (1985). Lambda lifting: Transforming programs to recursive equations. In Jouannaud, J.-P., editor, *Second Functional Programming Languages and Computer Architecture (Nancy, France)*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203. Springer Verlag.

[3] Louden, K. (1997). *Compiler Construction, Principles and Practice*. PWS Publishing.

# Index