Course Script

INF 5110: Compiler construction

INF5110, spring 2024

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 Introduction

The chapter covers different aspects of the **run-time environment** of a language. The run-time environment refers to the design, organization, and implementation of how to arrange the memory needs of a running program and how to access it at run-time. One way to understand the purpose of RTEs is: they have to *maintain abstractions* (concerning "data") 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 addressed adequately. "Abstractions" that need to be taken care of include variables inside scopes. Taking care means ultimately that code must be generated for that. In the simplest case, for a global variable in the global scope, access to that variable is code is simple, like a `load` or `store` instruction with a **static** address. If, on the other hand it's a local variable inside a procedure, the access to variables is not so simple any longer. Indeed, if the procedure calls itself in some form of recursion, one cannot speak about "the" variable, say x, to be accessed. There are multiple "versions" of the variable, one for each active call. And the compiler needs to arrange for that sitation, at least in a language that supports recursion, and generate code that will find and accesses the correct "instance" of x.

That's only one aspect of the tasks of the run-time environment, there are more. One needs to arrange for static and dynamic memory allocation, **parameter passing**, and **garbage collection**. The most important control abstraction in languages is that of a "**procedure**". Connected to that is the run-time **stack**, which is a part of the dynamic memory. The design of the run-time stack takes a good portion of this chapter.

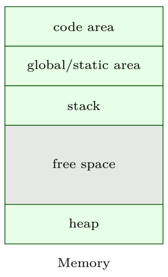


Figure 8.1: Typical memory layout

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 are known at compile time (and place in the part called static area), and dynamic means, they are not.

The static memory contains data for 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 the 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, wich leads to self-modifying code. That’s seen as a no-no, and, as said, measures are taken to prevent that. The generated code is not only kept immutable, it’s also treated mostly as static (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 is placed 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 or procedural 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**. Returning means that control continues at the point where the call originated.[1]. This call-and-return behavior is at the core of realizing the procedure abstraction. Calling a procedure involves executing a jump (`JMP`) and likewise the return is nothing else than executing an appropriate jump instruction, jumping back where one came from originally. Executing a jump does nothing else than setting the so-called program pointer to the address given as argument of the instruction. In the typical arrangement from the picture, the addresses jumped to are supposed to be located in code segment. Jumps themselves are therefore rather simple things, in particular, they are *unaware* of the intended call-return discipline: The jump address related to calling a procedure may be statically known (sometimes also not). The jump-address related to a return is **not** statically known.

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 ultimately, 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 of the implemented programming language, which for instance involve some form of parameter passing.

That involves the data area of the memory (since the code area is kept 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 the return address and use it for the jump back).

Calls can be *nested*, i.e., a function being called can in turn call another function. In that situation, procedure calls or **procedure activations**, are executed in *LIFO* fashion: the procedure called last is returned from first. That means, we need to arrange the remembered return addresses, one for each procedure activation, in the form of a stack, the **call stack**.[2] The run-time stack is a key ingredient of the run-time system. 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): 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 simplifications are possible, and one could do without an official dynamic stack data structure (though the call/return would still be executed under LIFO discipline, of course).

[1]Maybe not exactly at that point or line of code, but the line "immediately afterwards".

[2]There may be complications to the LIFO discipline, like exceptions. Or the languages supports something undisciplined as gotos, maybe even gotos that allows to jump out from a procedure call to "somewhere". That's nowadays seen as undesirable, and the lecture does not discuss it and problems it entails. Neither do we cover exceptions.

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**. That 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 as arguments upon being called. Indeed, the real power of the procedure abstraction does not just rely on code (control) being available for repeated execution, it owes its power on equal parts to the fact that it can be executed variously with 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, it needs to be arranged that at run-time the parameter passing is done properly. We will discuss different parameter-passing mechanisms later (the main ones being call-by-value, call-by-reference, and call-by-name, as well as some bastard scheme of lesser prominence). One complication is that when calling a procedure, the body may contain variables which are *not* introduced locally, 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 depends on the scoping rules of the language (**static** vs. **dynamic binding**).[3]

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 records** since the call of a procedure is also known as its activation. The chapter will discuss different ingredients and variations of such activation records, depending on features of the language. See Figure 8.2 for a schematic impression of an activation record. So in summary: when a procedure is called, at the time of the procedures **activation**, the return address plus further information needs to be stored. This packaged of data to be remember upon activation is called an **activation record**. Since calls and returns follow a LIFO discipline, those activation records are arrange in a **stack**, the call stack, which is an important part of the run-time environment. Another name for activation records is a **frame**. When arranged as part of a stack, another name for activation records is also **stack frame**.

Remark 8.1.1 (Non-stack frames?)**.** We said that frames, when arranged on the run-time stack, are also called stack frames. Does that mean, there are also frames or activation records *not* arranged on the stack? Indeed, there are languages where the stack arrangement is too restrictive! The last-in-first-out discipline underlying a stack reflects the way calls and returns behave during run-time. The return addresss, a crucial part of frames, certainly behave as stack, that's the core of calling and returning. Frames, however, contain more than just the return address, they contain also the local memory needs of a

[3]The chapter about symbol tables showed code examples for that, though the discussion was concerned with how symbol tables handle that, not how the run-time system deals with it. In the symbol-table coverage, some solutions for static binding made use of so-called **static links**. The concept of static links will reappear later in this chapter, for run-time environments.

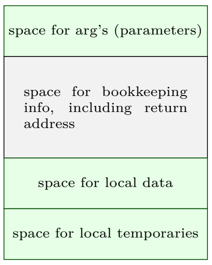

space for arg's (parameters)
space for bookkeeping info, including return address
space for local data
space for local temporaries

Figure 8.2: Schematic activation record

procedure activation (local variables and parameters). There are situations where the local data of a procedure activation is needed *after* the procedure has returned, i.e., after the activation of the procedure has ended: the procedure has returned but its data lives on! So the memory needs of an activation "survives" the call itself (the terminology is that qsome local variables **escape**).

How is that possible? In lexical scoping and if the language allows nested procedure definitions, an inner procedure definition can access the local variables of the surrounding procedure. If calling the outer procedure in turn calls the inner procedure, all is fine: the stack discipline means that the call of the inner procedure is terminated before the outer procedure. That means, the data from the other procure is available on the stack while the inner procedure runs: the life time of the activation of the outer procedure spans the life time of the activation of the inner procedure.

If, however, the outer procedure does not call the inner procedure itself, but **returns the procedure** to the caller for someone else to call it, then the outer procedure terminates with the inner procedure not yet activated and the activation record of the outer procedure is popped off the stack. If later, the inner procedure, the one that has been returned is called and activated, it has no access to local data from the outer procedure, since that has been popped off the stack. In other words: the activation records in a language with **higher-order functions** (and under **lexical** scoping) cannot be organized in a stack-based fashion. One still has activation records or frames (but not called stack-frames), and they will be allocated on the **heap.** □

Figure 8.3 shows the general impression of the code area or code segment. It is almost always neither moved nor changed at run-time and **statically** allocated, i.e., memory content representing machine code is not moved around nor changed or newly allocated at run-time.

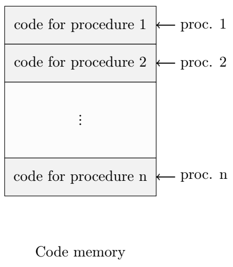


Figure 8.3: Code area

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 addresses given by a *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 plausible. Note that it's *not* the same as saying "the procedure called `P` occurring in the source code is located at such and such statically known 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 occurring 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.

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 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 shown schematically in Figure 8.4. A fully static scheme is rare for today's languages, but was the case for instance in old versions of Fortran (Fortran77). Nowadays, 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 Fortran in Listing 8.1.

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

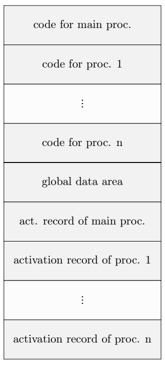

code for main proc.
code for proc. 1
⋮
code for proc. n
global data area
act. record of main proc.
activation record of proc. 1
⋮
activation record of proc. n

Figure 8.4: Full static layout

```
      CALL QUADMEAN(TABLE,3,TEMP)
      PRINT *,TEMP
      END

      SUBROUTINE QUADMEAN(A,SIZE,QMEAN)
      COMMON MAXSIZE
      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 don't matter too much.

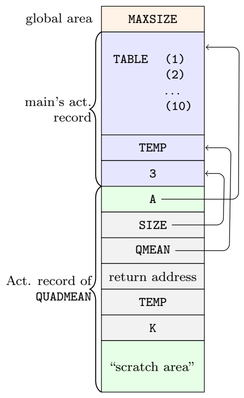


Figure 8.5: Static layout for the Fortran example

Important is the distinction 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 (mostly) organized in a stack, and then they are also called *stack frames* but it's the same thing. It's space used (at run-time) to fill the memory needs when calling the function (which is also known the function's activation). The necessary space involves slots for 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, i.e., in which exact order those pieces of information are arranged. Those details may also depend on the platform and the OS, not just the language being implemented. 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 run-time environments

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

Practically all full-scale programming languages allow recursion, and therefore need 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. The stack is also called **call stack** or **run-time stack**. As always, the exact format of the activation records depends on language and platform. More importantly, the rules how and where procedures can be defined and what one can "do" with them is different for different languages. 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 8.2 earlier. They have space for passing arguments, for local variables and so-called temporaries. The gray part in Figure 8.2 was said to be for book-keeping, administering 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, and at the bare minimum, it includes the return address.

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 (see also Remark 8.1.1).

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 achieve), not dynamic.

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

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

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

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

One step further, in the following section, will be 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 form of nesting will require to introduce, besides dynamic links, also static links (aka access links).

Remark 8.2.1 (Static link)**.** The mentioned notion of static links is basically the same as the one encountered before, when discussing the design of **symbol tables,** in particular how to arrange symbol tables properly for nested blocks and lexical binding. Here (resp. shortly later down the road), the static links will serve an analogous 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.

```c
#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.

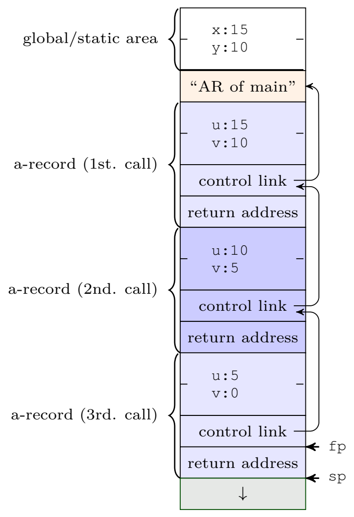


Figure 8.6: Stack for the gcd example (called with 15 and 10)

Besides the return addresses, the picture illustrates the notion of **control** or **dynamic** links, the **frame pointer**, and the **stack pointer.** The control links are also called dynamic links.

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`.

Conceptually, the stack consists of stack frames (3 in the picture), and each stack frame consists of a number of slots with information. The frame pointer points to the current activation record, i.e., the top-most frame in the stack of frames. The stack pointer points to the top-most slot on the stack that contains information, demarcating the border between the used and the unused parts of the stack memory. So the frame pointer identifies a stack frame, the top-most one, and it does so by poiting to *definite* position inside the activation record. It's logically possible to choose the definite possition of an activation record to be the end of the frame or record. In that case, the stack pointer and the frame pointer would point to the same address, as the end of the top-most frame is at the same time the end of the stack-memory currently in use. As indicated in Figure 8.6, this is typically not done. Instead, the frame pointer points to a well-chosen place somewhere in the middle of the frame such that some information belonging to the frame comes before that place, and some comes afterwards. The point is chosen in such a way, that the local data (variables, etc.) can be accessed fast, relative to the frame pointer, and that the pushing and popping of stack frame is likewise efficient. Note that access and the stack manipulations are done at run-time, so a design that allows efficient manipulations will not make the compiler as such more efficient, but results in faster code. What here is called vaguely "manipulations" of the stack here will later be discussed in more detail as so-called *calling conventions* which are basically the steps taken to build up the individual slots of a stack frame during calling and parameter passings (= push of a stack frames), resp. popping it off again upon returning.

Popping off a whole stack frame means, to make the second top-most frame the now top-most one. That means, one has to adapt the stack pointer and the frame pointer. Now, if the stack frame were uniform in size, that task would be simple, just increase[4] the address of stack and frame pointer but the fixed, uniform offset. Obviously, in general, activation records are *different in size*, as different procedures have different memory needs (a different amount for formal parameters and local variables, of different types). Even if different in size, frames are uniformely designed, with particular kinds of informations with known, best fixed, offsets from the frame pointer.

How to deal with the fact that frames are non-uniformely sized? That's exactly the purpose of the **dynamic** link or **control** link! It's a slot in a stack frame which points to the previous, older frame on the stack. Since frames are designed to have a definite "anchor" point from which a frame is being access, exactly the address pointed at by the frame pointer for the top-most stack, it's natural that the dynamic link identifies the previous stack frame by pointing to that frame's anchor point.

Alternatively, one could imagine a design, where instead of a dynamic link pointing to the previous stack, the stack frame contains information about its *size* and then calculating how much stack memory needs to be freed. However, using a dynamic link just pointing to (the anchor of) the previous frame is more efficient (and that's why we discuss the concept of dynamic links). In particular it's more efficient if the fixed anchor inside a frame, i.e., the point where the frame pointer and the dynamic links point to **is** the slot that contains the dynamic link itself: the frame is designed in such a way that the control link is located with an **offset of 0**. As a consequence, the control-links form a chain of addresses, the control-link in the top-most frame, pointed at by the frame pointer, points to the control link of the frame below the top-most one, etc., and the control link of a frame in this way remembers the frame pointer of the caller (at the time when the caller's frame was current and on top of the stack. That is shown in Figure 8.6. And popping off the top-most frame is setting the frame-pointer to the address contained in the slot the frame-pointer points at (which is control-pointer). That's fast, in particular when supported by hardware.

The sketched design choices, including the placement of the control link at an offset of 0 to the achor point of the frame is very common. Also plausible is that the return address is kept in close and fixed neighborship to the slot with the control link, here in the subsequent slot. Likewise common is that formal paramaters appear at positive offsets (higher up in the picture) and (other) local variables appear with negative offsets. It has to do with the fact that during parameter passing it makes sense to push the parmeters first on the stack, before proceeding adding the book-keeping informations (pushing the control link and the return address) and finally allocating space for the local variables of the callee. In this way, the order in which things are pushed on the stack (following the so-called calling conventions) reflects the layout of the stack frame (or vice versa). In the concrete gcd-example, the procedure has no additional local variables, i.e., the return address is located in the last slot of a stack frame.

Besides the parameters, there can be more local variables: C allows to introduce local variables, besides the formal parameters, in functions or procedures. What is not allowed

[4]That the address increases, instead of decreases depends of course on how the stack is layed out in memory. One convention is, that it grows towards lower addresses, and that is indicated in the pictures here in that the top of the stack is shown at the bottom.

is to introduce local procedures. There is still another general kind of data for which 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.

Side remark 8.2.2 (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 simple 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. □

Activation trees 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.

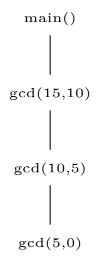
Example 8.2.3 (Activation records)*.* The code of Listing 8.3 contains some artificial code, which will use to illustrate the concept of activation trees (and a bit later also for another illustration of activation records and stack frames.

```
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)
```

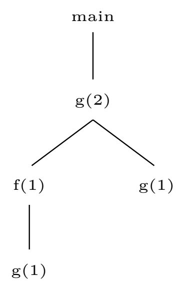
```
        { f(y);
          x--;
         g(y);
        }
    }

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. 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.[5]



(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 functions 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 activation 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.

[5]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 about parameter passing and various evaluation strategies later.

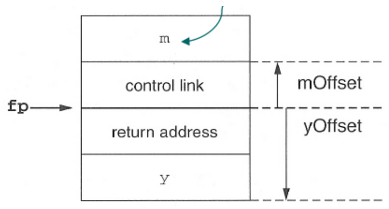


Figure 8.8: Layout `g`

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 `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.[6] Different presentations may employ different graphical conventions. The graphical conventions are of course to be distinguished from the layout itself of the activation record and the corresponding calling conventions (see later).

In Figure 8.8 and with the mentioned 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, i.e., the control link is kept at 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 pointer 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"[7] 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 described is a common and plausible one. It has to with the desire to build up the activation records in an efficient and clean way. When executing a call at run-time, a new activation record is **pushed** to the stack. But that is not an 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; 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

[6]In some pictures later, we let them point also to the "middle" of a slot.
[7]Confusingly, the top of the stack is written at the bottom.

layout of the activation records, are called a language's (or platform's) calling conventions (see later).

Layout for arrays of statically known size Procedures can of course declare local data more complex than data of basic or elementary types. The code from Listing 8.4s 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)`.

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.

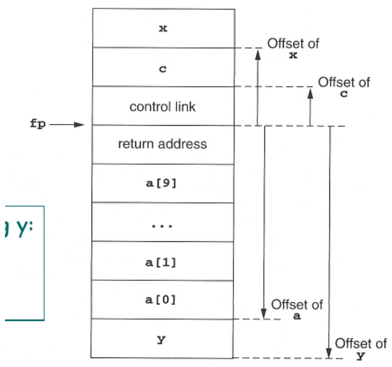


Figure 8.9: Layout

```
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 plausible assumptions on the size of the involved data: Addresses count 4 words, the character 1, the integers 2 words, the double 8. Syntax like `4(fp)` is meant 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 the very early day, hardware supports different 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 procedure involves pushing an activation record and returning means popping off an activation record. This is a view of 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 push the relevant information step by step to the stack, placing them at the designated places inside the activation record.

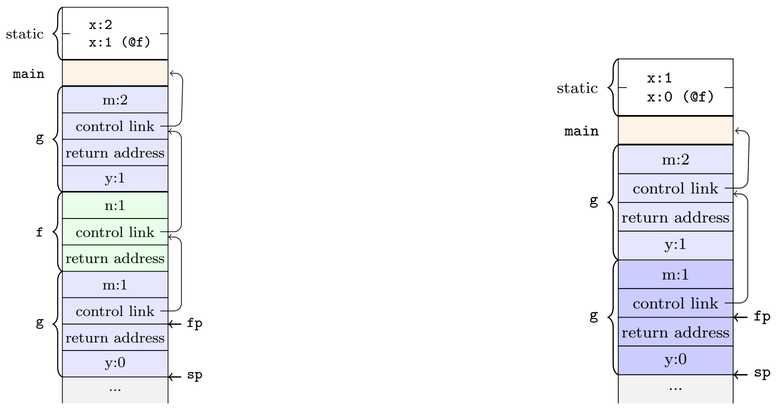
Someone has to perform those steps; the corresponding instructions have to be executed each time a function is called and "undone" when returning from a call. Since that happens at run-time, so those steps are not *not executed by the compiler*. But the compiler resp. compiler writer has to arrange for that they are 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 steps resp. the specification of those steps are called **calling sequences**, or also **linking conventions** or **calling conventions**.

Example 8.2.4. Back to the C code again from Listing 8.3 with the functions f and g. Figure 8.10 shows two snapshots of the stack. The stack from 8.10a shows 4 activation records. It corresponds to the point at run-time, when 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 stackpointer separates the used part of the stack from the free one (in gray). In the used part of the stack, there 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, the figures show 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 situation after the call to f, and g has again



(a) (b)

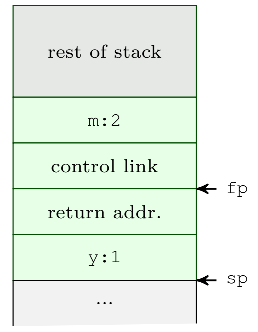
Figure 8.10: 2 snapshots of the call stack

be called. That corresponds to the right-most branch of the activation tree from Figure 8.7b. □

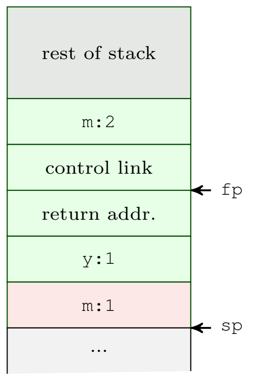
- For procedure call (entry)
  1. compute arguments, store them in the correct positions in the *new* activation record of the procedure (pushing them in order onto the runtime stack will achieve this)
  2. store (push) the `fp` as the *control link* in the new activation record
  3. change the `fp`, 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 8.2.5 (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. □

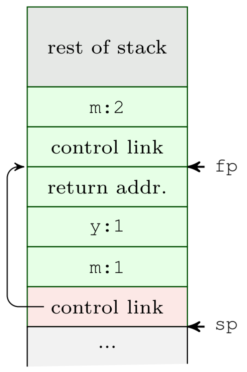
Treatment of auxiliary results: "temporaries" As explained, activation records contain space for various kinds of data, including space for local variables, including values for the formal parameters of the function in question. There is one part of the local memory need we have not mentioned yet. The code of the method body will in general do some calculation. At souce code level resp. in abstract syntax, that involves **compound expressions.**



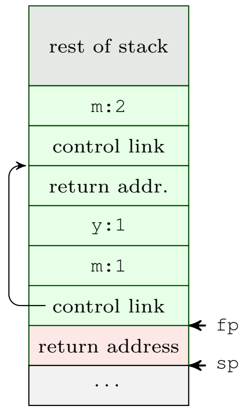
(a) Before the call



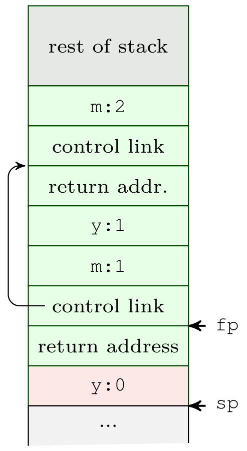
(b) m pushed



(c) `fp` pushed



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



(e) alloc. local var `y`

Figure 8.11: Steps when calling g

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, typical intemediate code does not support compound expressions. Instead, intermediate code breaks down compound calculation into their basic steps and uses additional local variables to store intermediate results. Those are known as temporary variables, or **temporaries**.[8] It’s one task of intermediate code generation to introduce those additional variables, i.e., generate intermediate code that makes 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

[8]The use of temporaries is at least done for so-called two-address and three-address codes. There exists also intermediate code formats that don’t use temporaries, but also they break down complex expressions into basic ones. In the chapter about intermediate code generation, we will see both flavors of intermediate codes.

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.[9]

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

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

Listing 8.7: Compound expression

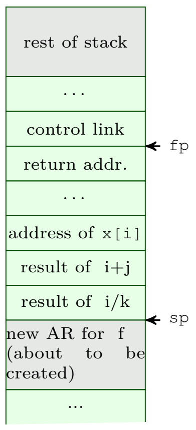


Figure 8.12: Temporaries

The computations in the example are not really complex from a 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 oraries. Next we shortly discuss variable-length data, concretely variable-length arrays.[10] Ada is a language that supports such data structures.

```
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

[9]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.

[10]Dynamic arrays also exists, and are even more flexible and nowadays more common. They can be handled the same way as described here.

Here, we illustrate how to deal with the situation that an variable-length array is passed as argument, in particular **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 pass the whole array as value.

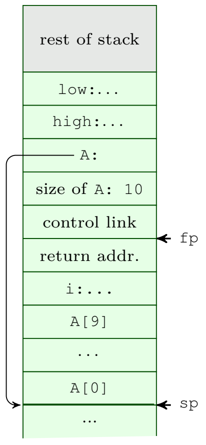


Figure 8.13: AR layout

The treatment is actually unproblematic; Figure 8.13 shows a possible layout . The picture simply says: if an array is passed as argument, the type may not specify its size, because only when passing the concrete array, the size is known. Then one just has to 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`.

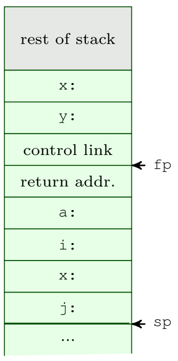
Nested declarations Before moving on to the more complex situation of nested procedure declarations, let's have a look at how do deal with blocks or scopes inside a procedure.[11]

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

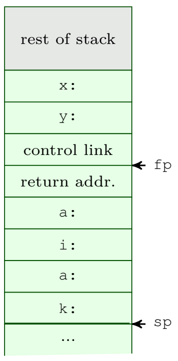
Listing 8.9: Nested declarations

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

[11]Some statement(s) enclosed by { and } is also called **compound statement** in the context of C [4].



(a) Area for block A allocated



(b) Area for block B allocated

Figure 8.14: Steps when calling *g*

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. In that way it's treated in the same spirit as union types in C. 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" of variables: global ones, which are static, and local ones which reside in the current stack frame. Languages with nested procedures (but without higher-order functions) are called **Pascal-like** languages.

With nested procedures (and 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 by **static links**.

The code from Listing 8.10 is not just Pascal-like, it's some concrete Pascal dialect. The comments after the `begin` and `end` statements indicate to which procedure that part belong). As `q` is nested in `p`, and as `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 `q`'s body 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. When procedure q from Listing 8.10 accesses the variable n, that refers to n declared in procedure p under **lexical** scoping.

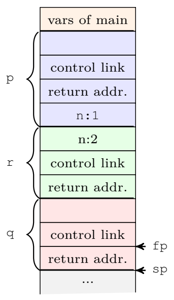


Figure 8.15: Stack frames (general) for calls m → p → r → q

This lexical nesting of the scopes is not reflected one-to-one in the *run-time* call stack. Fortunately, though, the call-stack and the lexical nesting are not completely independent. Variables defined locally in a procedure, like n in p, can of course be accessed inside p, including by procedures defined locally in p, like q (though in the shown code, q 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, 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 static links **multiple** times.

Figure 8.16 shows the same sitation as the run-time stack from Figure 8.15, this time filling out more details. A crucial addition is of course, the **static links** indicated in red.

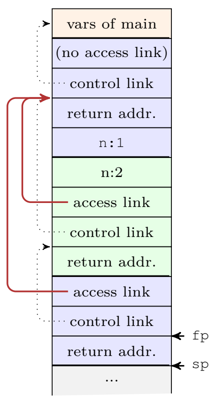


Figure 8.16: Static (or access) link

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 scope.

Access links, same as control links point "to" a stack frame. As explained, the point of reference is neither the start nor the end of the frame; the "anchor point" of the stack frame is the where the frame pointer 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 link.

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`.

Side remark 8.2.6 (Calling procedure `r` outside of `q`?)**.** The explanation claims that `r` cannot be called from outside of **q** in the example. To be precise, that's not 100% true, though not shown in the example. As seen earlier, Pascal supports **function variables**. With those, it's possible, to pass a locally defined procedure to the outside, so that it can be accessed and called from there. We will look at the consequences of that in the following section, when discussing higher-order functions. The current example and the current section is not concerned with that more complex setting, it's only about nested procedure definitions, not higher-order procedure and/or procedure variables. □

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 times (or in general multiple times), a phenomenon called **access chaining.**

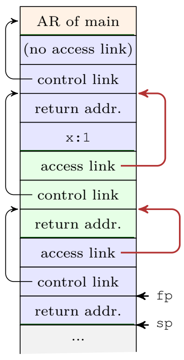


Figure 8.17: calls m → `p` → `q` → `r`

Using dot-notation to access slots inside an activation 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 an 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 to generate an appropriate access chain with the chain-length statically determined, but 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:

$$\texttt{fp.}\underbrace{\texttt{al.al.al.}\ldots\texttt{al}}_{n}\texttt{.x}$$

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 chain. It's assumed that the static link is contained at 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, 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 on the problem and the personal programming style, but still, nesting of functions comes easy in functional languages.

As noted earlier: a stack-based run-time environment 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 activation records on a stack.

Now, that the activation records have 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 adjusting `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**

Figure 8.17 illustrated lexical nesting involving more than just one nesting level, which required access chaining. There is another aspect we have mentioned only in passing.

The access link points to an activation record corresponding to the lexically enclosing procedure. But of course the enclosing procedure could have been called multiple times in recursive situations, 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? It's probably clear: it's the "most recent" stack frame. That's illustrated in Figure 8.18.

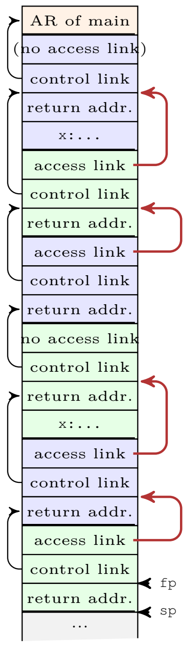


Figure 8.18: `main` → `p` → `q` → `r` → `p` → `q` → `r`

Functions as parameters, closures, and higher order functions

There is more to scoping and run-time environments than 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 have seen a Pascal example with **procedure variables**, which we will revisit here. We also shortly mentioned in connection with static links, without a concrete example involving procedure variables and without going into details, that such variables complicate matters. Ultimately, when dealing with full higher-order functions, one cannot arrange the activation records on a stack.

Of course, also for higher-order functions, the calls and returns follow a LIFO discipline. So, there is still a notion of a call-stack. The stack in the run-time system is not just there to manage the return addresses 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). However:

A stack arrangement for the data needs 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, and when 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 the call stack in the way dicussed.

For higher-order function, this alignement of the life-times of function activations and data declared in functions is no longer given. Same if one has variables containing (pointers to) functions, as in the Pascal example. Therefore, one needs 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's called **closure**.

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

Pascal supports procedure parameters (and local procedures), but cannot give back procedures via "return", so it's not a higher-order language. See Listing 8.13.

```
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 thus we can still stay within a stack design 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 of closures. 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 “closures are heap-allocated”. In general, that’s the case, but, as said, we are restricting ourselves to procedures as parameters, which allows to work with a stack. So 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 (but most common) semantics of how 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. Activation records can access variables which are neither local nor handed over officially as formal parameters. So for those one normally does not speak about parameter passing, 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.

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 referred to as `a` in the body of `p`. That variable is 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 stored 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.

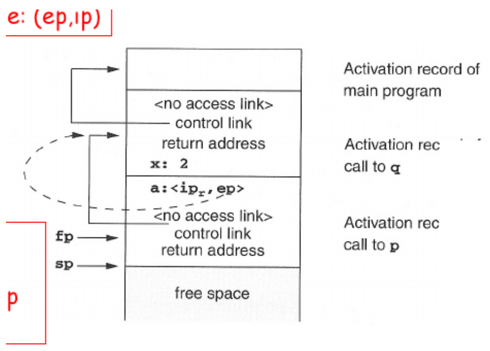
To be able to cal `r` (via `a` in the source-code), not only the address needs to be known, but also the "lexical nesting situation" of the called procedure, so that we can fill the **static link** slot in `r`'s activation record properly. At static time, that is not known, i.e., it's not known which 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 8.19a, 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.

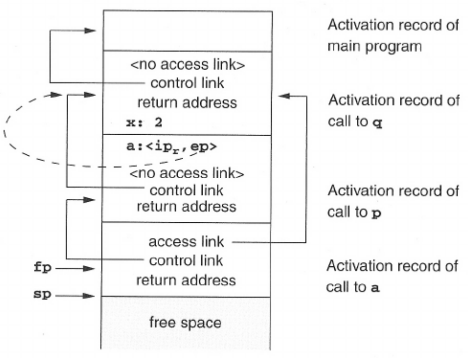
This pair represents the **closure**!

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. Calls and returns follow a LIFO strategy anyway, i.e., show a stack-like behavior, and that is also supported by hardware, in the form of providing a 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 some book-keeping information, depending on the language, to make the scoping work. The return addresses need a stack, that's for sure, but that one can also put the dynamic memory need in



(a) Closure for formal parameter a of the example



(b) After calling a (= r)

Figure 8.19: Stack-allocated closures

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 **cannot outlive** the procedure's activation.

As long as that is the case, stack allocation of frames is fine. But there are situations where the data is needed **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 is 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 (a reference to) 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. Data could also be "returned" via 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 stored (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 return 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".[12] 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 references creating an object in the `dangle` procedure and then returning 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.[13] Goto's easily break any scoping rule, 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)

[12]Of course the part of the memory is still "there", perhaps even, if lucky, the latest value of `x` is still there. But the memory may also have in the meantime been reused for a different activation record with different data. At any rate, it should be considered an "illegal" access.

[13]Starting end of the 60ies, and in the seventies, there were the so-called structured programming wars. The first important salvo in these was were one of Dijkstra's notes, the particular one titled "Go To statement considered harmful".

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
  $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.[14]

The Pascal code 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.

```go
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 local to `f` and uses `x`, which is non-local to `g` but local to `f`, and is being return from `f`. One also say, it's a situation when `x` **escapes** its scope.

In languages supporting full higher-order functions, functions are treated as "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" in the language (there may even be more classes). 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 as arguments. Or, as in C-like languages, they cannot be declared in a nested way, something possible for ordinary data. At any rate, they are treated in a more restricted manner. For higher-order functions, there are no such restrictions. Like all (other) data, functions can being locally defined, are allowed as arguments and as return values for

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

other functions. To manage the memory needs for such language, one needs a so-called **fully-dynamic** run-time environment. For that one needs **heap-allocated closures**, i.e., more flexibel ones than those stack-allocated closures, but conceptually, closures have the same function as before. Memory management in general 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 (or actually maybe not). Using side effects on functional variables is not central to a language like Pascal, and if used, it should be used with care. However, returning functions and (re-)combining 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, the management is part of the run-time environment, and thus covered in this chapter. 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 "garbage collection" is typically not used to refer to that part of the memory managent in the run-time environment.

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. We discussed that calling a procedure means going through some well-defined sequence of steps (called the calling sequence among other things) filling in information into a new activation record, and those steps include handing over the arguments from caller to callee.

Parameter passing is the mechanism by which the caller and the callee communicate, via appropriate slots in their activation records. The communication is bi-directional in the general case: the caller hands over the arguments to the callee at call-time and, upon returning, the callee can hand back a result.

Let's focus for a start on the input parameters of a called procedure, not the return. 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).

One aspect, however, was not really discussed, namely what exactly is passed from caller to callee (and back), and how. Sure, the "parameters" are passed, but there are **different**

ways to do that. Two basic alternatives are: make a **copy** of the value to hand over, or alternatively hand over a **reference** to the slot where the caller keeps the value, such that the callee can access it. This latter way, using a reference not a value, is more obvious for the calling a procedure. For returns, the callee cannot just return the reference to a slot where it stores the value to be returned; 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 focuses on calling, not returning).

8.3.1 Call by-value, by-reference, and by-value-reference

Call-by-value is conceptually probably the simplest and clearest. Parameter passing (when calling) 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, but it can 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. A variable of a class type or of an array type does not contain the object itself or the array itself, but a reference or pointer to the data (typically on the heap).

Now assume to call a function with (a reference to) an object or array as argument, or references in general. If we assume a language with call-by-value and 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 mean, 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 "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 one would not need a special word for that: all data is passed uniformely by by-value, and that includes references.

Does it matter? It depends, perhaps it's a bit splitting hairs, especially from the perspective of the user of a language. Passing a reference or a reference data in a 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 version 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 to 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 , one cannot not use the call-parameters for (also) communicating results back to the caller (if we ignore call-by-value-reference). For that, one has a `return` statement. But that's not the only way. One finds also languages which support *two* kinds of parameters, parameters for calling, as usual, and parameter(s) for returning. They are sometimes called **in-parameters** and **out-parameters.** In such a setting, a procedure declaration specifies 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 treated in 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, 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.

Procedures or functions may operate with variables in their 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 (closures). Independent from how complex the language design is wrt. procedures, there are *two* kinds of variables whose values originate from *outside* the procedure itself. One are of course the input parameters, 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 parameters", the variables that come from somewhere outside?

For the global variables, they are of course not copied, their address is globally known. 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 the 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 λ-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, a prominent example of an imperative, procedural language using call-by-value. Listing 8.19 and 8.20 shows two 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. 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`.

In C and Java (and many other languages), call-by-value is the only parameter passing method. Some language insist that formal parameters are **immutable**. Of course, if one passes by value a reference to a piece of data, the variable itself may be immutable, but it does not disallow changing the content of the referenced data. The code from Listing 8.20 is an example for that. 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 a 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.[15]

[15]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 ...

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

Listing 8.21: An array as argument

Side remark 8.3.1 (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 explicitly 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, 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);
        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, which is parameter passing. One is that there are two methods called `inc2`. Depending on whether the method is called with `x1` (an `int`) as argument or `x2` (an `Integer`), 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 then is 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 were 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 not much to do 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 ++ operation and some conversions 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 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 typed by `Integer`. An expression like `(new Integer(5))++` is illegal; as said, ++ works on variables only. In particular ++ is *not* interpreted as 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 (but they do have a "get-method", called `intValue`).

If, however, ++ is applied to a variable of type `Integer`, 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 as 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 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 operates on the formal parameter `x` of type `Integer`, which contains a copy of a reference, at least at the beginning. 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) and the callee mutates instance variables in that object, then of course the calleer will see those changes. But for the special case of `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. . .[16]

```
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 8.20.

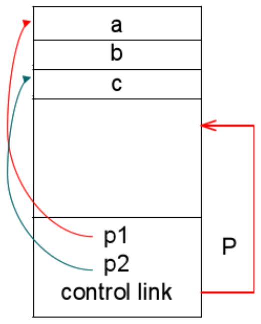


Figure 8.20: Call-by-reference, activation record layout

Call-by-value-result

As said, call-by-value and call-by-result are the two main alternative for procedural, imperative languages. But there exist also lesser known alternatives. One is **call-by-value-result**. As said, 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 returning the content of the formal parameters is hand back to the caller. Some languages, like Ada, are quite explicit about that fact, in that the distinguishe between `in` and `out` parameters. In contrast, the strategy here is not explicit about that, it just has **one kind** of parameters, but they are used in a two-way form of 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 C 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

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

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. For instance: **when** are the value of actual variables determined when doing "actual ← formal parameters". Is it done when calling? Or 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[17].

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

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. When calling a function with an argument, it's not the value that is handed over, but 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. It's therefore corresponds to **delayed evaluation**. If the argument is needed more than once, it needs to be *recalculated* over and over again, which can degrade performance (unless remedied by a technique called memoization).

A possible way to implement that scheme is the following. In 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. One may even be tempted to say: don't try to understand it, it's just a

[17]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.

brain teaser. Just avoid programming that way, you will only shoot yourself in the foot . . .

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 the evaluation of the argument **has side-effects.** 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 support is passing procedures as arguments. 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 times. 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 8.26 shows a function with one integer argument. Additionally, the changes the argument, incrementing it:

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

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 8.26: Call-by-name example

That's not the only side effect the procedure does, it additionally increments `i` in Listing 8.26, 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 8.27 is basically analogous, and perhaps a little less artificial. It uses a procedure called `swap`, whose body executes a typical way of swapping the content of two variables, here containing integers, corresponding to the formal parameters `a` and `b`. Swapping 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 reason 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 8.27: Swapping

Applying the swap-procedure under call-by-name may run into problems: Applying it to the the pair `i,a[i]` does *not* swap 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,[18] 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 apart 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**.

[18]Actually also under dynamic scoping.

So, we silently assume capture-avoiding substitution or assume that the swap uses `j` 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 8.27[19]

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

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

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

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 8.28 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 8.28: 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 discussion may give the impression that call-my-name is more than a bit weird. It may perhaps be historically interesting and used in languages like Algol 60[20], but otherwise the evolution of programming languages left behind call-by-name as a bad idea, forgot about it, and the only place where it occasionally shows up is in academic compiler construction or programming language courses, discussing it as curiosity.

[19]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.

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

A valid point, for imperative languages. Combining side effects and delayed evaluation leads to behavior hard to understand, error prone, and with no obvious upside. If a 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 matters 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 expression”[21] 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 evaluated. In other words, all that can get much more confusing than invoking `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** those 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 a function now or later will give the same result. Indeed, no matter how many times it will get evaluated, it always gives the same result (for the same input). Call-by-value corresponds to a scheme, where 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’s called memoization.

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

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

The answer comes from that fact that it lazy and eager evaluation do **not** lead to 100% 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

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

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 infinite lists, not *actually* infinite lists, 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 at once **would not terminate**, and there's not enough memory to store an infinite list anyway.[22] 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 procedure (a stream of integers, actually, the infinite *Fibonacci series*.

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

Listing 8.29: Lazy evaluation/streams

The first line indicates the type of the function `fib`, taking two numbers (the "seed" of the series, typically 1 and 1) and returning an element of `[Int]`. That's a list, more precisely in Haskell a lazy list, i.e., a stream. The generation is done it the recursion case in the last line of the snippet, where the colon-operator `:` appends `m` to the rest of the stream, generated by the recursive call to `fib`.

8.4 Virtual methods in object-oriented languages

Next, we shed some light on aspects of the run-time system relevant for object-oriented languages. Not too much light, though, and basically only for one aspect for some mainstream object-oriented languages, like C^{++} or Java (and a bit on Smalltalk). 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 their 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 words virtual method, late bound method and dynamically bound method interchangably.

Connected to this concept is also **dynamic dispatch**. Dispatch is what the run-time system has to do when calling a procedure, function, method etc., i.e., *jumping* to the beginning of the code of a procedure body. Determining the jump target, i.e., locating the code of the procedure, can be done at compile time or at run-time. That's **static dispatch** resp. **dynamic dispatch** (and it corresponds to static binding resp. dynamic binding of the name of the procedure, method, function . . .). If the compiler can determine

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

the jump-target, then it can produce code with the jump address "hard-coded" into the jump-command, which is more efficient. 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 procedure names only. The association of `x` with its address is likewise called *binding*. We have mostly looked a statically or lexically bound variables. For instance, using static links to locate the proper address of a statically bound variable in languages with nested procedures.

The concept of late-bound method is **central** object-orientation (class-based or otherwise). It just means the binding is done at run-time and done dynamically not statically. The concept is so characteristic for methods, that one might get the impression that it's *the* feature that distinguishes methods from procedures or functions from non object-oriented languages. Both methods and procedures/functions are "callable units" with parameter passing, i.e., comparable languages concepts. In non higher-order languages, procedures are indeed statically bound (and handled by static dispatch). Of course, there are many languages that support higher-order functions. Even an old-school language like Pascal, which is not higher-order, features procedures as parameters (and references to procedures). Having formal parameters for functions means, that there are situations (maybe in the body of the function with a functional parameter) where statically its unknown which concrete function will be called. In other words, it's a situation of dynamic binding and dynamic patch. It's only that no one calls it late binding of function, as it's just the standard *parameter passing* and that of course always "binds" dynamically the actual parameters to the formal ones.

Let's have a look at the code from Listing 8.30.

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

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

class C: public B {
  public:
  double u;
  virtual void h();
};
```

Listing 8.30: Inheritance, virtual and non-virtual methods + fields

The code sketches a situation with class inheritance showing an inheritance hierarchy with three classes. The classes implement virtual and static methods, as well as some fields.

The interesting part are the virtual or late bound methods, `g` provided in classed `A` and `B`, and `h` provided in classes `B` and `C`. The example also shows non-virtual, static methods, namely `f` in `A` and `B`.

For the static methods, the situation is called **shadowing** (not overriding). We have encountered a Java-example for that earlier in the lecture. This part of the code will be less relevant for the discussion here, as we are focussing on how to realize late-binding.

Assume we have three instances for the three classes. The objects are allocated on the heap, and Figure 8.21 sketches their layout. The figure focuses on the fields, as we will different design choices how to represent method only later. In Figure 8.21, information concerning methods is still missing, as we will different representational alternatives later.

The identifiers `A`, `B`, and `C` from Listing 8.30 are, in languages like C^{++}, Java etc., at the same time the names of classes that are used to create 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 (names of) static types

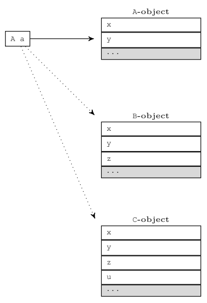


Figure 8.21: Layout

So `a` in the figure is a variable of static type/class `A`, and containing a reference to an instance of class `A`. Now `B` is subclass of `A`, and `C` a subclass of `C`. That means, `C` is a subtype of `B`, which in turn is a subtype of `A` (which also implies that `C` is a subtype of `A`). I many types languages that support class inheritance, subclassing implies subtyping. What does subtyping exactly means? It's a form of **polymorphism**, as discussed in the chapter about typing: variables or pieces of data of some type have also as further types all supertypes. Here, an object of static type `C` also has types `B` and `A`. In a type-safe language this is analogous to say that at places where an object of static type `A` is allowed, also objects of `B` or `C` are welcome, without causing problems, i.e., all fields an methods that are needed of an `A`-typed object are offered by `B` and `C`-objects. That's assured by the fact that inheritance (in it's conventional form) only **adds new members** (or **overrides** i.e., replace existing ones with a new implementation). In figure 8.21, the dotted arrows indicate that `a` with static type `A` can point also to instances of `B` or `C`, as mentioned.

Table 8.3 gives an overview, which code is executed when invoking a method with a particular name on an instance of which class. So, `a` is meant as as variable with a `static` type `A`, `b` of static type `B` ....

The columns on the left shows the situation for the non-virtual, static method `f`. For `f`, the code that is executed is determined by the static type. More interesting (and more compex) is the situation for `g` and `h`. For instance, when calling `a.g()` (where `a` is assumed to be of static type `A`, the `a` can refer to an instance of `A`, `B`, or `C`. Said differently, the **run-time type** of the object `a` refers to can be `A`, `B`, or `C`. With late-binding, it's not the static type that determines which code is executed, but the object. That corresponds to the standard mental picture, that an object is an entity consisting of “data, i.e., the fields, together with procedure that operate on the data, the methods”. Both fields and methods together are also known as the object’s *members*. So `a.g()` can refer to the `g`’s code fom class `A` or form the code from class `B`, if *a* contains an instance of `B` or `C`. Since `C` does not override `g`, instances of `C` operate with `g` inherited from `B.` For the virtual method `h` the situation is similar.

call	target	call	target	call	target
`a.f`	`A::f`	*a.g*	`A::g` or `B::g`	`a.h`	illegal
`b.f`	`B::f`	*b.g*	`B::g`	`b.h`	`B::h` or `C::h`
`c.f`	`B::f`	*c.g*	`B::g`	`c.h`	`C::h`

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

So far a recap of what late-binding of methods has to achieve. Now to the correspinding arrangement in memory, and how to realize the dispatch; for late-bound methods, dynamic dispatch.

As mentioned, objects are allocated on the heap (Figure 8.21), and contain the content for the fields. . Their representation frames or activation records, which is not surprising as they serve an analogous purpose: provide space for the “local information”. For objects, the content of the fields and for frames the content of local variables. Frames also have additional information for book-keeping (return addresses, dynamic links . . .) which is missing from objects. Missing, however, is information about the objects’ methods. We discuss two arrangements for late-bound methods.

For static methods, the object representation on the heap does not contain any information, as the instances have nothing to do with static methods resp. static methods have nothing to do with instances. In particular, the body of `f` from our example has **no access** the instance variables like `x`.[23] For non-virtual methods like `f`, the corresponding code is statically know, and the dispatch code for calls like `a.f()` involves a staticically determined jump-address, based on its the static type. So the missing information in the objects in Figure 8.21 only refers to virtual methods. The obvious solution would be to simply add enough slots into the objects, so that every member is represented: not only slot for every field, but also one slot for (a reference to) every virtual function. That is known as *embedding*, but many conventional object-oriented languages chose different representations. We discuss two prominent solutions, virtual function tables and relying on a representation of the inheritance relation between classes.

8.4.1 Virtual function table

The layout based on virtual function table for the example is sketched in Figure 8.22. Each object points to one its virtual function table, which in turn contain the information where to find the corresponding code. For instance the instance of `C` at the bottom supports the virtual methods named `g` and `h`, where the code for `g` is inherited and the one for `h` provided by `C`, in a situation of overriding.

[23]Unless the variable would be declared as `static` (in Java) as well, and then represented as “part” of the class. Indeed, invoking notationally the static method `f` “on” an instance as in `(new A().f());` feels misleading and writing `A.f()`, which is possible in Java, may seem clearer.

That looks just like an additional layer of indirection (which it is), compared to embedding the corresponding pointers directly into the objects. The advantage is that it allows for a compact, uniform design with **statically fixed off-sets.** The slot for the virtual function table is placed at a known offset inside the object's memory.[24] And the individual tables place methods uniformely at fixed offsets. That's shown in the figure that `g` is placed at offset 0 not just in the instance of class `A`, the top-most class, but at the same offset for instances of all its subclasses. Analogously for method `h`, which is placed at offset 1 in instances of `B` and `C` and further potential subclasses. This allows to generate code for a dynamic dispatch for calling `g` with commands that correspond to `jmp a.virttab[g_offset]`: get the object referenced in `a`, then to the slot pointing to |its virtual function table, with a fixed offset, and then another statically known offset to locate the proper code for `g`.

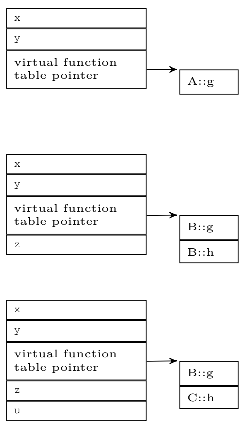

x
y
virtual function table pointer
A::g
x
y
virtual function table pointer
z
B::g
B::h
x
y
virtual function table pointer
z
u
B::g
C::h

Figure 8.22: Virtual function table

If placing the pointers to the code section uniformely and with fixed offset is such a good idea, why place them in external virtual function tables instead of doing the same with slots inside the object, avoiding one layer of indirection? One could do that, but the price of a uniform layout with **fixed** offsets would be that one had to reserve fixed slots for **all** methods throughout the program maybe including libraries, so there would be (empty) slots in an object reserved for methods not even supported and thus never called.

8.4.2 Less disciplined and/or more flexible object-oriented languages

Virtual tables are a good data structure to realize dynamic method dispatch for C^{++} or similar languages. There exist, however, differently designed object-oriented languages with different features. Common for all object-oriented languages are certainly objects and late binding of methods. Some of the features known from C^{++} or *Java* may be missing, or realised differently, and as a consequence, virtual function tables may no longer be practical or possible. One important, object-oriented language is Smalltalk,[25] predating C^{++} and

[24]That need not be the first slot, as anyway only objects with virtual methods need a virtual function table anyway.
[25]One should mention that there exist many flavors of Smalltalk.

influenced by Simula, the first object-oriented language. One feature that distinguishes Smalltalk from C^{++} or *Java* is that it lacks a static type system. In a way, Smalltalk is a kind of "object-oriented Lisp", in contrast to C^{++} which is an "object-oriented C". Also, (most versions of) Smalltalk makes no real distinction between classes and objects, resp. classes are also objects. Another related complication is that "classes" or objects in Smalltalk **can be extended** by adding a new method. Smalltalk is not the only language that allows to add at run-time new methods to objects (or other changes). Sometimes such languages are called *object-based* instead of object-oriented.

Anyway, such flexibility comes as a price. It's no longer possible to have the efficient arrangment where the run-time system can realize the dynamic dispatch with statically known fixed offsets. Instead, the run-time system needs to consult the class-hierarchy and **search** for the correct method, resp. its code to dynamically dispatch to it. For instance, for the code from Listing 8.30 we use before, when calling `g()` on an instance of C, the search follows the links from classes to super-classes until the method is in, in the example in the class called `C`. Smalltalk, however, is not statically typed. So unlike the situation in statically typed languages like Java or C^{++}, it may happen that one invokes a method on an object that is not offered, in which case an run-time (type) error is raise. In the picture, the case when a method cannot be located is indicated by the undefined "super-class" $\perp$.

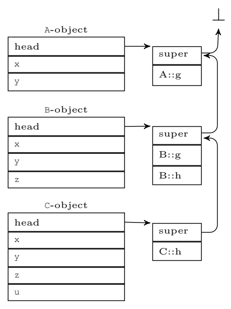


Figure 8.23: Late-binding à la Smalltalk, ...

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).[26] We discussed the stack at some length, but remarked also which kind of data is allocated on the heap, basically dynamic data whose lifetime is not aligned with the activations of procedures. That may be reference data (like objects) but also activation records in a language with higher-order functions.

Dynamic memory means that it's allocated and deallocated at *run-time*. There are different alternatives for that, including that it's left to the programmer, i.e., the allocation is done at program-level with specific commands ("alloc", "free"). That can be efficient, but is seen as error prone. **Garbage collection** is a part of the run-time environment

[26] Just in case someone wonders: the notion of heap here unrelated to the well-known heap-data structure from algorithms & data structures.

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 off an activation record frees a segment from the stack memory and reclaims the corresponding portion for reuse. One could call it therefore a form of garbage collection and it certainly is a form of automatic memory management, but normally it’s not called garbage collection.

Compared to the stack, memory allocation on the heap does not follow a LIFO discipline that stems from calling and returning from procedure calls. Automatic memory allocation and deallocation on the heap is less disciplined, more complex and less efficient.

The heap contains all all dynamically allocated data that cannot be managed by the stack. That consists of all data whose life time can exceed the lifetime of the procedure call that allocates the data. This concerns typically data that is *“pointed to”*, i.e., where the creator keeps a pointer or reference to the created data, not the value itself. The pointer can be returned when an activation ends, which means the pointer is *copied* whoever called the procedure, and the callee activation record can be deallocated. That frees also the space containing the original pointer, but not the space of the data being pointed at. Since the callee has gotten a copy of the pointer, the data is still accessible, it’s not **garbage**, and thus the data needs to be created outside the stack, on the heap.[27]

As a consequence, the heap typically contains: **records**, **objects**, **arrays**, all of which are mostly reference data structures. For languages with higher-order functions and lexical binding, also their memory needs need to be represented at the heap: the memory needs of procedures that are *returned* from another procedure outlive the latter procedure’s activation. We have also discussed what happens in Pascal, which supports function pointers and some (not so elegant) way to give access to a nested procedure from the outside without allocating the corresponding piece of memory on the heap. The (erroneous) situation is analogous to the dangling-reference mentioned above.

To do automatic garbage collection, the run-time environment need to figure out, at run-time, 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 arithmetic as such are unproblematic. 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. In such a setting, it’s not unreasonable to expect from the programmer to clean up unused memory. After all, it’s hard to have it both ways: low-level access and

[27]Earlier in this chapter, Listing 8.16 showed an example where the callee gets hold of an address on the *stack* (using C’s address operator `&`). That’s an “illegal address” and known as *dangling reference*. Proper garbage collection (on the heap) makes sure that on the heap, there are no dangling references: if some piece of program points to a cell on the heap, it’s by definition no garbage and the reference does not dangle . . . Of course, garbage collection and pointer arithmetic as in C don’t live in easy harmony, standard C does not have a garbage collector.

free manipulation of memory *and* support of the compiler to protect oneself to misuse the freedom.

For higher-order functions, the situation is different. Functions are a powerful **abstraction.** Programming with higher-order procedures can lead to complex access patterns on the heap and in the address space. Higher-order function would basically be unusable if one offers the user those abstractions, but at the same time expects that the user cleans up the activation records, which would require low-level understanding of what's going on. It 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 with some basic design decisions As part of the run-time system, garbage collection is done at run-time. Still, it works **approxiatively**, 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 only approximatively is of course, that it cannot precisely foresee the future, even if it tried. Neither the compiler nor the running program can 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, and one basic decision is whether it involves **moving** objects during the clean-up or not.[28] By moving an object, we mean, copying the content from one place in the heap to another, better suited one and of course adapt according the addresses that mention the object.

Never moving objects (in that sense) will lead to **fragmentation** of the heap space. One may end up with many stretches of unused space, many of which so small that they are unusable for larger objects. This way 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 collector can move non-garbage objects. There are different ways to do that and we look at two variations of that, known as **mark-and-sweep** and **stop-and-copy**.

Another design decision is to **when** to do garbabe collection. On obvious answer here might be: "do it only when really needed". Collectors that kick in when the program runs out of memory are also called **batch collectors.**

[28]Let's call data pieces in the heap with the generic name "object", even if the data represents records, closures or other heap allocated data, not just objects in the object-orinented sense.

Alternatives may be to do it somehow "continuously", always cleaning up "a bit". Of course, there is also a trade-off between having an overly eager garbage collector and one that only kicks in when the memory runs full: doing it agressively, where the garbage collector kicks in after each couple of steps of the program or in very short intervals to try to hunt down possible new pieces of garbage may be inefficient.

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

Generally, garbage collection is based on the following observation: heap addresses are only **reachable** in the following two ways: either **directly** through variables.[29] or **indirectly** following fields in reachable objects (or other heap-located forms of data), which point to further objects ... In this way, the heap forms 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. Marking basically does a **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 then 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 knows all the potentially used objects. Implicitly one has determined also all definitely unused ones, the garbage, namely all unmarked one. Now, having determined the garbage as the pool of unmarked objects, however is not in itself useful, as one does not have the information readily at hand.

That's where the second phase comes in, the **sweep**. The garbage collecter goes through the heap again, this time sequentially, i.e., no 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 new object it grabs a slot from free list.

Afterwards, there is, 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 adjusting pointers.

[29]The variables may have their allocated memory in the static part of the memory, or on the stack, but may also and additionally have their value kept in registers.

Stop-and-copy

Another form of garbage collector is is known as **stop-and-copy.** It can be seen as improvement over the mark-and-sweep type of garbage collectors. Here, the space for heap-management is split into **two halves**, and only one is in use at any given point in time. So one half contains the heap data including garbage, the second half is unused. From time to time, namely when the garbabe collector has done its jobs, the two halves flip their role: the garbage collector stops the program, marks the heap data similar as in the mark-and-sweep method, and copies the non-garbage to the unused half, freeing thereby the previously used one. When copying the used data, it's of course compacted in the process and all pointers and addresses have to be adapted, same as before.

The marking phase now does not have to use an extra bit to label pieces of data to be treated in a subsequente sweep phase. The collector can copy the data during one recursive pass through the graph, and by copying it into the free half, doing compaction at the same time.

The "stop"-part of the stop-and-copy terminology hints at a problem with this and also the mark-and-sweep style of garbage collection: the garbage collector from time to time stops execution of the user program to do its clean-up. In interactive applications, not to mention those with real-time requirements, that is problematic, as the program freezes in unpredicable intervals.

The "stop-the-world" kind of garbable collection have been improved, to avoid that a program has to pause intermittently. There are "generational" garbage collectors concurrent garbage collector, whole books have been written on engineering garbage collectors [3], but it's beyond the scope of this lecture.

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] Jones, R. and Lins, R. (1996). *Garbage Collection – Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons Inc.

[4] Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition.

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

Index