Course Script

INF 5110: Compiler construction

INF5110, spring 2021

Martin Steffen

Contents

Chapter
Run-time environments

What is it about?

Learning Targets of this Chapter

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

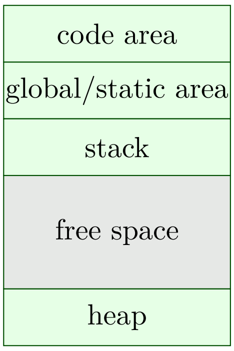
Contents

8.1 Intro

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

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

8.1.1 Static & dynamic memory layout at runtime



Memory

typical memory layout: for languages (as nowadays basically all) with

- static memory
- dynamic memory:
  - stack
  - heap

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

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

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

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

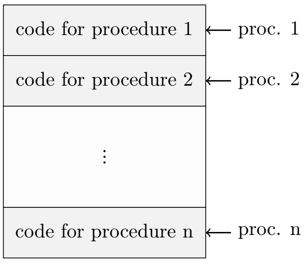
So far we have discussed only the bare bones of the run-time environment to realize the procedure abstraction (the heap may be discussed later): in all by the very simplest settings, we need to arrange to maintain a stack for return addresses and manipulate the stack properly at run-time. If we had a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

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

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

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

8.1.2 Translated program code



Code memory

- **code segment**: almost always considered as **statically** allocated

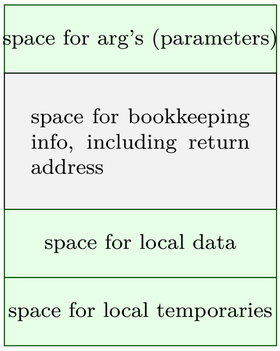
⇒ neither moved nor changed at runtime

- compiler aware of all addresses of "chunks" of code: *entry points* of the procedures
- but:
  - generated code often *relocatable*
  - final, absolute adresses given by *linker* / *loader*

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

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

8.1.3 Activation records


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

Schematic activation record

- *schematic* organization of activation records/activation block/stack frame ...
- goal: realize
  - parameter passing
  - scoping rules /local variables treatment
  - prepare for call/return behavior
- *calling conventions* on a platform

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

8.2 The procedure abstraction: different layouts

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

8.2.1 Static layout

A full static layout of the run-time environment means, that the location of “everything” is known and fixed at compile time. All addresses of all of the memory known to the compiler, for the executable code, all variables, and forms all forms of auxiliary data (for instance big constants in the program, e.g., string literals). Such a layout is schown schematically in Figure 8.1. A fully static scheme is rarely the case for today’s languages, but was the case for instance in old versions of Fortran (Fortran77). Nowayday, there coud be special applications, where static layout is used, like like safety critical embedded systems.

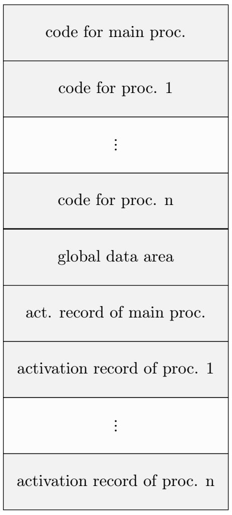

code for main proc.
code for proc. 1
⋮
code for proc. n
global data area
act. record of main proc.
activation record of proc. 1
⋮
activation record of proc. n

Figure 8.1: Full static layout

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

```
      PROGRAM TEST
      COMMON MAXSIZE
      INTEGER MAXSIZE
      REAL TABLE(10),TEMP
      MAXSIZE = 10
      READ *, TABLE(1),TABLE(2),TABLE(3)
      CALL QUADMEAN(TABLE,3,TEMP)
      PRINT *,TEMP
      END
      SUBROUTINE QUADMEAN(A,SIZE,QMEAN)
      COMMON MAXSIZE
      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 8.2 matter not too much.

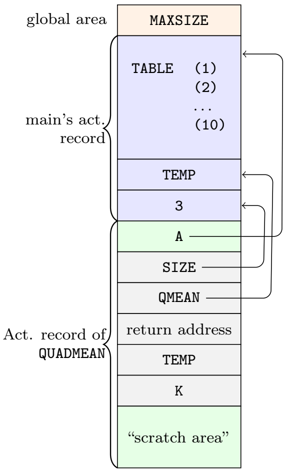


Figure 8.2: Static layout for the Fortran example

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

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

8.3 Stack-based runtime environments

8.3.1 Stack-based runtime environments

- so far: no(!) *recursion*
- everything's static, incl. placement of activation records
- *ancient* and *restrictive* arrangement of the run-time envs
- calls and returns (also without recursion) follow at runtime a LIFO (= **stack-like**) discipline

Stack of activation records

- procedures as *abstractions* with own *local data*

⇒ run-time memory arrangement where procedure-local data together with other info (arrange proper returns, parameter passing) is organized as stack.

- AKA: *call stack*, *runtime stack*
- AR: exact format depends on language and platform

8.3.2 Situation in languages without local procedures

- recursion, but all procedures are *global*
- C-like languages

Activation record info (besides local data, see later)

- *frame pointer*
- *control link* (or *dynamic link*)[1]
- (optional): *stack pointer*
- *return address*

Where we are dealing with languages that support recursion, but no nesting of procedure declarations. A prominent example for that is C, therefore we say we study activation records for C-like languages. 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 nesting will require to introduce, besides dynamic links, also static links

The notion of static links mentioned in the footnote is basically the same we encountered before, when discussing the design of symbol tables, in particular how to arrange them properly for nested blocks and lexcical binding. Here (resp. shortly later down the road),

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

the static links serve the same purpose, only not linking up (parts of a) symbol table, but activation records.

8.3.3 Euclid's recursive gcd algo

```
#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;
}
```

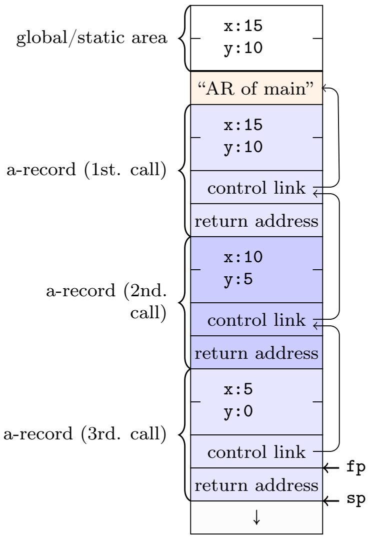
Not that it's the focus of the example, but the C-code represents a simple recursive implementation for calculating the greatest common divisor of two integers (making use of some modulo calculation in the recursive call, that's the `%` operator). C is also uses call-by-value a parameter passing mechanism.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

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

It's a simply form of recursion, also in connection with run-time environments. The call which pushes a new stack frame to the run-time stack, is the last thing that happens in an activation. That means, the space of the caller's stack frame and the local data it contains therein, is not actually needed any longer. That means, one could arrange the run-time environment in such a way, not to add another stack frame for the callee, but to recycle the space of the caller's frame. If one (resp. the run-time system) still make 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.

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; expect of course, a return to the code calling from the outside needs to be done, in the example the return to the `main` method.

8.3.4 Stack gcd



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

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

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

The exact design of activation record may vary. The need for a concept like the control link comes from a simple fact. [More here. . .]

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

8.3.5 Local and global variables and scoping

Code

```c
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;
  }
```

- global variable `x`
- but: (different) `x` *local* to `f`
- remember C:
  - call by value
  - static lexical scoping

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

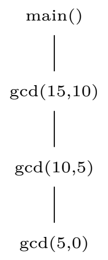
8.3.6 Activation records and activation trees

- *activation* of a function: corresponds to: *call* of a function
- **activation record**
  - data structure for run-time system
  - holds all relevant data for a function call and control-info in "standardized" form
  - control-behavior of functions: LIFO
  - if data *cannot* outlive activation of a function
  - ⇒ activation records can be arranged in as **stack** (like here)
  - in this case: activation record AKA *stack frame*

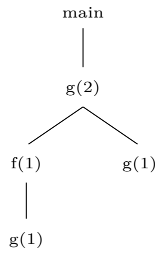
The two pictures illustrate the notion of *activation tree* (where in the gcd-case, is not much of a tree as it's linear. An activation of gcd calls itself at most once (actually, gcd is tail-recursive).

8.3.7 Activation record and activation trees

GCD

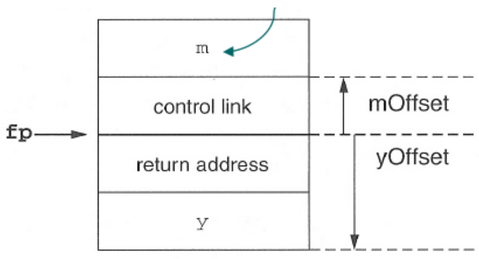


f and g example



8.3.8 Variable access and design of ARs

Layout g



- fp: frame pointer
- m (in this example): parameter of g

Possible arrangement of g's AR

- AR's: structurally *uniform* per language (or at least compiler) / platform
- different function defs, different size of AR

⇒ *frames* on the stack differently sized

- note: FP points
  - not: "top" of the frame/stack, but
  - to a well-chosen, well-defined position in the frame
  - other local data (local vars) accessible *relative* to that
- conventions
  - higher addresses "higher up"
  - stack "grows" towards lower addresses

The pictures use the following concvention. They show "pointers" or arrows to point to the "bottom" of the meant slot. For example, `fp` points to the control link which has offset 0 from that pointer. The return address given the slot below has a negative offset to that pointer. Different presentations may employ different graphical conventions. The graphical conventions are of course to be distinguished from the "calling conventions" and the design of the activation record. One agreement in this layout is: 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.

8.3.9 Layout for arrays of statically known size

Code

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

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

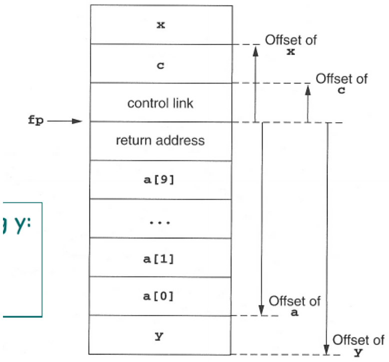
access of `c` and `y`

```
  c: 4(fp)
  y: −32(fp)
```

access for `a[i]`

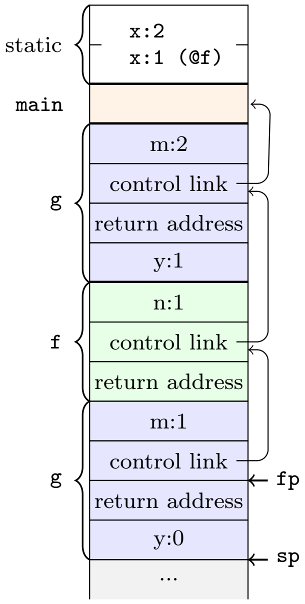
```
  (−24+2∗i)(fp)
```

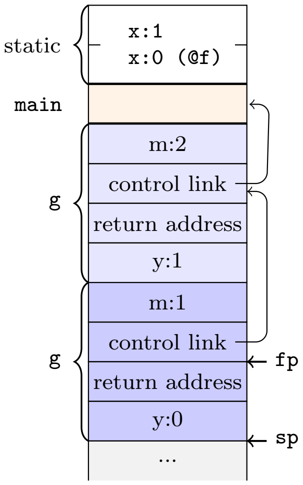
Layout



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

8.3.10 2 snapshots of the call stack





- note: call by value, x in f *static*

The picture on the slide refers to the simple, artificial C example involving procedures f and g, which was also used to illustrate the activation tree.

8.3.11 How to do the "push and pop"

- **calling sequences**: AKA as *linking conventions* or *calling conventions*
- for RT environments: uniform design not just of
  - data structures (=ARs), but also of
  - uniform *actions* being taken when calling/returning from a procedure
- how to *do* details of "push and pop" on the call-stack

E.g: Parameter passing

- not just *where* (in the ARs) to find value for the actual parameter needs to be defined, but well-defined **steps** (ultimately **code**) that copies it there (and potentially reads it from there)

- "jointly" done by compiler + OS + HW
- distribution of *responsibilities* between caller and callee:
  - who copies the parameter to the right place
  - who saves registers and restores them
  - ...

8.3.12 Steps when calling

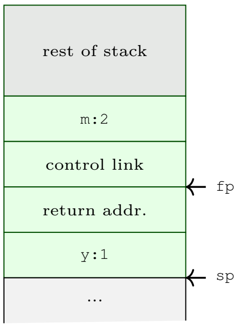
- For procedure call (entry)
  1. compute arguments, store them in the correct positions in the *new* activation record of the procedure (pushing them in order onto the runtime stack will achieve this)
  2. store (push) the fp as the *control link* in the new activation record

    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

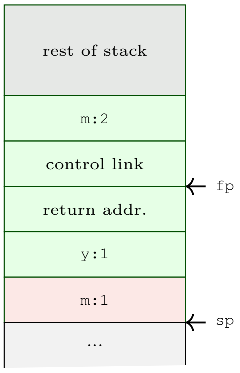
8.3.13 Steps when calling g

Before call



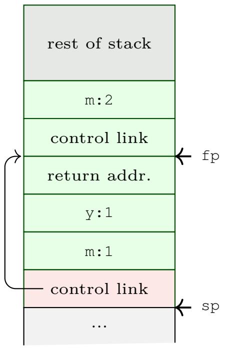
before call to g

Pushed m



pushed param.

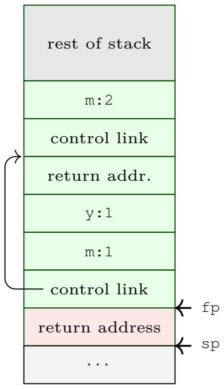
Pushed fp



pushed fp

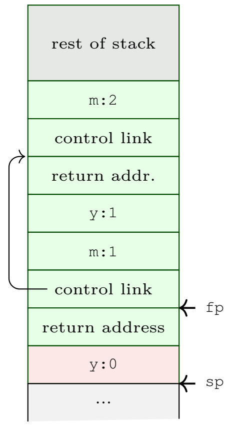
8.3.14 Steps when calling g (cont'd)

Return pushed



fp := sp,push return addr.

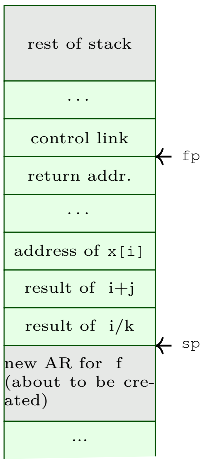
local var's pushed



alloc. local var y

8.3.15 Treatment of auxiliary results: “temporaries”

Layout picture



- calculations need *memory* for intermediate results.
- called **temporaries** in ARs.

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

- note: `x[i]` represents an *address* or reference, `i`, `j`, `k` represent *values*
- assume a strict left-to-right evaluation (call `f(j)` may change values.)
- *stack* of temporaries.
- [NB: compilers typically use **registers** as much as possible, what does not fit there goes into the AR.]

The array example uses arrays indexed by integers. Integes are good (efficient) for array-offsets, so they act as “references”. In a way, calculations like that is a form of pointer arithmentic. That, however, is not the message of the slide. The message of the slide is, that the body of a procedure may involve more complex operations than elementary additions etc. The computations in the example are not really complex from programming perspective, but they are *compound*. Perhaps there may be hardware support for `x + y`, `x-y`, `x+1` etc, but compound expressions are of course not natively supported. They have to be broken down to elementary calculations and the intermediate results need to be stored somewhere. The memory entities for those intermediate results are called *temporaries*. We will encounter them when talking about *code generation* (where we need to generate code that breaks down compound expressions into indivual steps). That comes later, for the run-time enviromenent, the design of the activation record must provide enough space so be able to locally store those results.

The side remark says, that often, one tries to avoid putting all local temporaries inside the activation record, as much as possible, one would like to use registers for that.

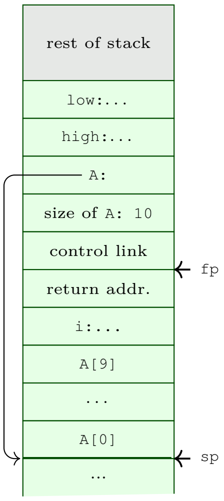
8.3.16 Variable-length data

Ada code

```
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;
```

- Ada example
- assume: array passed *by value* (“copying”)
- `A[i]`: calculated as `@6(fp) + 2*i`
- in Java and other languages: arrays passed *by reference*
- note: space for `A` (as ref) and size of `A` is fixed-size (as well as `low` and `high`)

Layout picture



AR of call to SUM

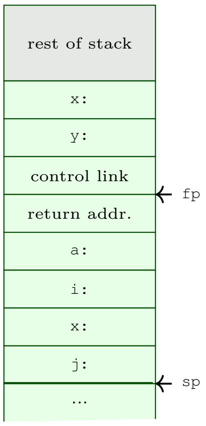
The picture and the slide simply says: if an array passed as argument is allowed to have a non-fixed size, that’s fine. When passing the array, the size is known, just store the size at one particular, agreed upon place in the activation record (here offset 6), and then used the value for your calculation when accessing a slot. So, compared to the previous handling of arrays, there is just one layer of indirection involved.

8.3.17 Nested declarations ("compound statements")

C Code

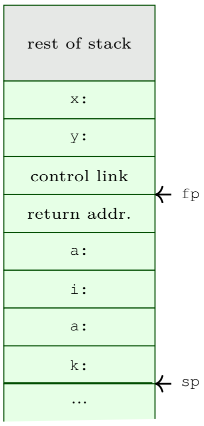
```
void p(int x, double y)
{ char a;
  int i;
  ...;
 A:{ double x;
    int j;
    ...;
  }
  ...;
 B: { char * a;
    int k;
    ...;
  };
  ...;
}
```

Nested blocks layout (1)



area for block A allocated

Nested blocks layout (2)



area for block B allocated

The terminology of *compound statements* seems not widely used, at least not in the sense used here. The gist of the example is: if one has local scopes of that kind (here called A and B, there is no need to allocate space for both (in that way it's treated in the same spirit as union types). The space for the local variables from the first scope maybe reused for the needs of the second. There is also no need to officially "push" and "pop" activation records following the calling conventions (though nested scopes do follow a stack-discipline and they could be treated as "inlined" calls to anonymous, parameterless procedures).

8.4 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, by which one mean: 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 are in the current stack frame. The local ones can be accessed by offsets from the frame pointer.

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

8.4.1 Nested procedures in Pascal

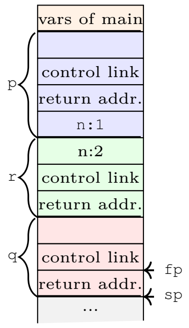
The code is in some form of Pascal. The comments after the `begin` and `end` statements indicate to which procedure that part belong. Since `q` is nested in `p`, and since `p` has a local variable `n` in the same scope, this local variable `n` is accessible inside `q`. At run-time, in an call to `q`,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

```
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.
```

- proc. `p` contains `q` and `r` nested
- also "nested" (i.e., local) in `p`: integer `n`
  - in scope for `q` and `r` but
  - neither *global* nor *local* to `q` and `r`

8.4.2 Accessing non-local var's

Stack layout



calls m → p → r → q

- `n` in `q`: under *lexical* scoping: `n` declared in `procedure p` is meant
- this is not reflected in the stack (of course) as this stack represents the *run-time* call stack.
- remember: static links (or access links) in connection with *symbol tables*

Symbol tables

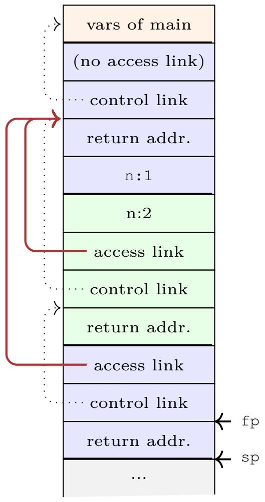
- “name-addressable” mapping
- access at compile time
- cf. scope tree

Dynamic memory

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

8.4.3 Access link as part of the AR

Stack layout



calls m → p → r → q

- **access link** (or **static link**): part of AR (at fixed position)
- points to stack-frame representing the current AR of the statically enclosed “procedural” scope

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

8.4.4 Example with multiple levels

```
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.
```

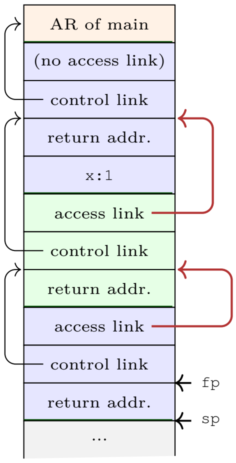
In the example procedure p contains procedure q and that in turn contains r. That is the static structure, which is relevant for lexically scoped variables. At run-time, the main procure calls p which calls q which calls r, so that order is somehow aligned with the static nesting structure, but that's 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.

Well, to be precise, it's not 100% impossible (though not shown in this example). As we have seen earlier, Pascal supports *function variables*. With that, it is in principle possible, to pass a locally defined procedure outside via the variable, so that it can be accessed that way from the "outside". We will look at the consequences of that in the following section, when we are discussing higher-order functions. So the current example and the current section is not concerned with that more complex setting, it purely about nested procedures here, 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.

8.4.5 Access chaining

Layout



calls m → p → q → r

- program chain
- access (conceptual): `fp.al.al.x`
- access link slot: fixed "offset" inside AR (but: AR's differently sized)
- "distance" from current AR to place of `x`
  - not fixed, i.e.
  - *statically* unknown!
- However: **number of access link dereferences statically known**
- lexical **nesting level**

8.4.6 Implementing access chaining

As example:

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

- access need to be fast => use registers
- assume, at `fp` 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
```

- often: not so many block-levels/access chains nessessary

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

Variable `x` is assumed at an offset of 6 in the frame that corresponds to the scope where `x` is defined.

Of course, following a chain of access links is costly. The slide optimistically states that realistically, the chains are not really very long in practices. That is, I guess, plausible, for instance for Pascal program. On the other hand, in languages where functions play a central role (i.e., in functional languages), a programmer may well structure the code with functions nested inside functions nested inside functions etc. Of course that depends a bit in the problem and the personal programming style, but still, nesting of functions comes easy in functional languages.

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

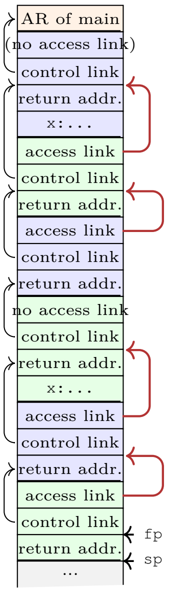
8.4.7 Calling sequence

- 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 runtume stack will achieve this)
  2. - **push access link**, value calculated via link chaining (“ `fp.al.al....` ”)
     - 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.
  1. store the return address in the new AR, if necessary
  2. perform a jump to the code of the called procedure.
  3. Allocate space on the stack for local var’s by appropriate adjustement of the `sp`
- procedure exit
  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**

8.4.8 Calling sequence: with access links

Layout



after 2nd call to r

- main → p → q → r → p → q → r
- calling sequence: actions to do the "push & pop"
- distribution of responsibilities between caller and callee
- generate an appropriate access chain, chain-length statically determined
- actual computation (of course) done at run-time

8.5 Functions as parameters

There is more to scoping and run-time environments that nested procedure declarations. We have seen glimpses of that before, mostly in the context of Pascal. We have seen it in the Pascal example with *procedure variables*. We will revisit that example here. We also shortly mentioned in connection with static links, without a concrete example involving procedure variables and without going into details, that those variable complicate matter. Ultimately, making in impossible to arrange the activation records on a stack, when dealing with full higher-order functions.

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

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

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

We start less ambitious, though, we don't fully embrace higher-order functions, but look at functions as parameters only. In that setting, the alignment of local data and function allocation *still* holds, though it get's more complex. Anyway, with this alignment one *still* can make a stack-based arrangment. In a way, one has *stack-arranged* closures.

However, generally, when talking about full closures, they are normally heap arranged. There are not many languages nowadays that bother to support procedure parameters without also support procedures as return values.

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

8.5.1 Procedures as parameter

```
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.
```

8.5.2 Procedures as parameters, same example in Go

```
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();
}
```

8.5.3 Procedures as parameters, same example in ocaml

```
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'' *)
```

8.5.4 Closures and the design of ARs

- [?] rather "implementation centric"
- closure there:
  - **restricted** setting
  - specific way to achieve closures
  - specific semantics of non-local vars ("by reference")
- higher-order functions:
  - functions as arguments *and* return values
  - nested function declaration
- similar problems with: "function variables"
- Example shown: **only** procedures as *parameters*, not *returned*

8.5.5 Closures, schematically

- independent from concrete design of the RTE/ARs:
- what do we need to execute the body of a procedure?

Closure (abstractly)

A closure is a function body[2] *together* with the values for all its variables, including the non-local ones.[2]

- individual AR not enough for all variables used (non-local vars)
- in *stack*-organized RTE's:
  - fortunately ARs are *stack*-allocated
  - → with clever use of "links" (access/static links): possible to access variables that are "nested further out"/ deeper in the *stack* (following links)

8.5.6 Organize access with procedure parameters

- when calling p: allocate a stack frame
- executing p calls a => another stack frame
- number of parameters etc: knowable from the type of a
- *but* 2 problems

"control-flow" problem

currently only RTE, but: how can (the compiler arrange that) p calls a (and allocate a frame for a) if a is not know yet?

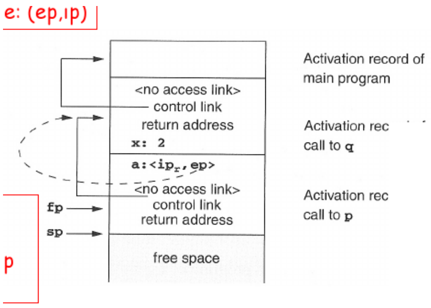
[2] Resp.: at least the possibility to locate them.

data problem

How can one statically arrange that a will be able to access non-local variables if statically it's not known what a will be?

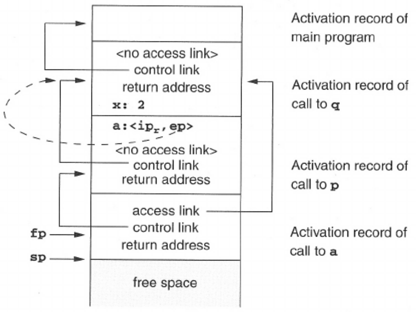
- solution: for a procedure variable (like a): *store* in AR
  - **reference** to the code of argument (as representation of the function body)
  - **reference** to the frame, i.e., the relevant *frame pointer* (here: to the frame of q where r is defined)
- this pair = **closure**!

8.5.7 Closure for formal parameter a of the example



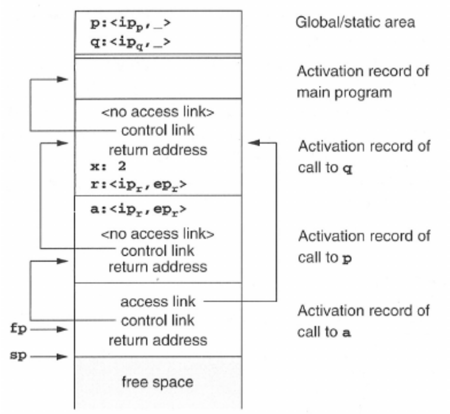
- stack after the call to p
- closure $\langle ip, ep\rangle$
- *ep*: refers to q's frame pointer
- note: distinction in calling sequence for
  - calling "ordinary" proc's and
  - calling procs in proc parameters (i.e., via closures)
- that may be unified ("closures" only)

8.5.8 After calling a (= r)



- note: *static* link of the new frame: used from the closure!

8.5.9 Making it uniform



- note: calling conventions *differ*
  - calling procedures as formal parameters
  - “standard” procedures (statically known)
- treatment can be made uniform

8.5.10 Limitations of stack-based RTEs

- procedures: **central** (!) control-flow abstraction in languages
- stack-based allocation: intuitive, common, and efficient (supported by HW)
- used in many languages
- procedure calls and returns: LIFO (= stack) behavior
- AR: local data for procedure body

Underlying assumption for stack-based RTEs

The data (=AR) for a procedure cannot **outlive** the activation where they are declared.

- assumption can break for many reasons
  - returning *references* of local variables
  - higher-order functions (or function variables)
  - “undisciplined” control flow (rather deprecated, goto’s can break any scoping rules, or procedure abstraction)
  - explicit memory allocation (and deallocation), pointer arithmetic etc.

8.5.11 Dangling ref’s due to returning references

```
int * dangle (void) {
  int x;        // local var
  return &x;    // address of x
}
```

- similar: returning references to objects created via `new`
- variable’s lifetime may be over, but the reference lives on . . .

8.5.12 Function variables

```
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.
```

```
funcvar
Runtime error 216 at $0000000000400233
  $0000000000400233
  $0000000000400268
  $00000000004001E0
```

8.5.13 Functions as return values

```
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)
}
```

- function `g`
  - defined local to `f`
  - uses `x`, non-local to `g`, local to `f`
  - is being returned from `f`

8.5.14 Fully-dynamic RTEs

- full higher-order functions = functions are “data” same as everything else
  - function being locally defined
  - function as arguments to other functions
  - functions returned by functions

→ ARs cannot be stack-allocated

- closures needed, but *heap*-allocated (≠ Louden)
- objects (and references): *heap*-allocated
- less “disciplined” memory handling than stack-allocation
- **garbage** collection
- often: stack based allocation + fully-dynamic (= heap-based) allocation

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.

8.6 Parameter passing

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

We also discussed typical designs of activation records, with the frame pointer as the "anchor" to the activation record. Besides other information, the activation records in particular contains space for the parameters of a procedure, if any. Parameters passing the mechanism where the caller "communicates" with the callee activation. That is bi-directional in the general case: the caller hands over the actual parameter values to the callee at call-time and, upon returning, the callee can hand back a return value.

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

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

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

8.6.1 Call by-value, by-reference, by value-reference

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

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

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

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

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

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

8.6.2 Communicating values between procedures

- procedure *abstraction*, **modularity**
- parameter passing = communication of values between procedures
- from caller to callee (and back)
- binding actual parameters to forma ones
- with the help of the RTE
- **formal** parameters vs. **actual** parameters
- two principal versions
    1. by-value
    2. by-reference

8.6.3 CBV and CBR, roughly

Intro

Core distinction/question

on the level of caller/callee *activation records* (on the stack frame): how does the AR of the callee get hold of the value the caller wants to hand over?

1. callee's AR with a *copy* of the value for the formal parameter
2. the callee AR with a *pointer* to the memory slot of the actual parameter

- if one has to choose only one: it's call-by-value
- remember: non-local variables (in lexical scope), nested procedures, and even closures:
    - those variables are "smuggled in" *by reference*

  - [NB: there are also *by value* closures]

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

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

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

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

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

The technique to systematically promote outside variables to official parameters is known as λ-lifting. It's mostly used in some compilers for functional languages ([**?**]).

8.6.4 Parameter passing by-value

The first `inc2` example does not work, of course, if the intention of the function is to do a double increment. Sure, the function increments its integer argument by 2, allright, 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 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, it would actually be allowed to do incrementations and other calculations on addresses or pointer; that's known as pointer arithmetic. Java and other languages would don't offer that. Some make the distinction in terminology that Java has references whereas C has pointers.

How is it in Java, needs clarification

The fact that some variables do not contain data values directly but a pointer to the place where to find the value is not visible directly to the programmer in Java. There is no need to explictly figure out the address of some place of data nor to explicitly dereference and address to obtain the value. That's all behind the scenes. In connection with parameter passing and this increment example: if one had a method `inc2`, then the declaration `void inc2 (int x) {x++;x++;}` would corresponds to the first C-example. And a declaration using type `Integer` for the parameter instead corresponds to the second example.

If one tries it out on real Java code, one may, however, get some suprise. See Listing 8.2.

```
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 8.2: Call-by-value or by-value-reference, or what?

There are some aspects of the code, unrelated to the issue at hand, namely parameter passing. One is that there are two methods called `inc2`. Depending on whether the method is called with `x1` as argument or `x2`, the appropriate one is chosen. The parameter for both versions is of different type, `int` vs. `Integer`, and that's good enough for disambiguation. That's an example of *overloading*, more precisely, of `method overloading`, a variant of `polymorphis`. We brushed upon overloading in the chapter about types and type checking. Another aspect crucial for parameter passing is the fact that the methods are `static`, the same would occur when using late-bound methods.

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

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

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

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

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

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

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

If, however, ++ is applied to a variable of type `Integer`, what then happens is that the object of type `Integer` is *converted* to the corresponding integer value of type `int`, and in that way `x++` in the second method is well-typed and works, with some conversion going on behind the scenes. This implicit conversion can be interpreted in leading to a form of *polymorphism*. The line between overloading and conversions of that kind is a bit blurred; both count among so-called *ad-hoc* polymorphism accordin to the seminal disucssion of different forms of polymorphism from [**?**]. In that paper, such conversions are called *coercions*.

That should make clear what happens in Listing 8.2. The reference to the integer object is passed by-value, the body operators on the variable `x`, the formal parameter of type `Integer`, which contains a copy of a reference, at least at the beginning. Operating on `x` doing `x++` does not change the state of the integer object, but creates a new one, to which the parameter `x` points, thereby severing the connection to the caller's reference kept in `x2`.

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

- in C: CBV only parameter passing method
- in some lang's: formal parameters "immutable"
- straightforward: *copy* actual parameters → formal parameters (in the ARs).

C examples

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

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

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

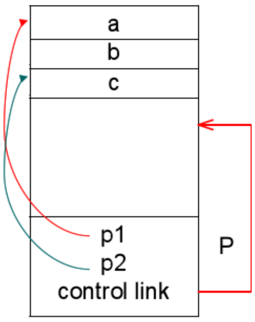
arrays: "by-reference" data

8.6.5 Call-by-reference

- hand over pointer/reference/address of the actual parameter
- useful especially for large data structures
- typically (for cbr): actual parameters must be *variables*
- Fortran actually allows things like `P(5,b)` and `P(a+b,c)`.

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

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



8.6.6 Call-by-value-result

- *call-by-value-result* can give *different* results from cbr
- allocated as a *local* variable (as cbv)
- however: copied “two-way”
  - when calling: actual → formal parameters
  - when returning: actual ← formal parameters
- aka: “copy-in-copy-out” (or “copy-restore”)
- Ada’s `in` and `out` paremeters
- *when* are the value of actual variables determined when doing “actual ← formal parameters”
  - when calling
  - when returning
- not the cleanest parameter passing mechanism around...

8.6.7 Call-by-value-result example

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

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

- C-syntax (C has cbv, not cbvr)
- note: *aliasing* (via the arguments, here obvious)
- cbvr: same as cbr, unless *aliasing* “messes it up”[3]

8.6.8 Call-by-name (C-syntax)

- most complex (or is it . . . ?)
- hand over: textual representation (“name”) of the argument (substitution)
- in that respect: a bit like *macro expansion* (but lexically scoped)
- actual paramater *not* calculated *before* actually used!
- on the other hand: if needed more than once: *recalculated* over and over again
- aka: *delayed evaluation*
- Implementation
  - actual paramter: represented as a small procedure (*thunk*, *suspension*), if actual parameter = expression
  - optimization, if actually parameter = variable (works like call-by-reference then)

8.6.9 Call-by-name examples

- in (imperative) languages without procedure parameters:
  - delayed evaluation most visible when dealing with things like `a[i]`
  - `a[i]` is actually like “apply `a` to index `i`”
  - combine that with side-effects (`i++`) ⇒ pretty confusing

Example 1

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

- call as `p(a[i])`
- corresponds to `++(a[i])`
- note:
  - `++ _` has a side effect
  - `i` may change in . . .

Example 2

```
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;
}
```

[3] One can ask though, if not call-by-reference would be messed-up in the example already.

8.6.10 Another example: “swapping”

```
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]);
```

- note: local and global variable i

8.6.11 Call-by-name illustrations

Code

```
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)
```

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

8.6.12 Call by name (Algol)

```
begin comment Simple array example;
  procedure zero (Arr,i,j,u1,u2);
  integer Arr;
  integer i,j,u1,u2;
begin
    for i := 1 step 1 until u1 do
      for j := 1 step 1 until u2 do
        Arr := 0

end;

integer array Work [1:100,1:200];
integer p,q,x,y,z;
x := 100;
y := 200
zero(Work[p,q],p,q,x,y);
end
```

8.6.13 Lazy evaluation

- call-by-name
  - complex & potentially confusing (in the presence of *side effects*)
  - not really used (there)
- declarative/functional languages: **lazy** evaluation
- optimization:

  - avoid recalculation of the argument
  - ⇒ remember (and share) results after first calculation (“memoization”)
  - works only in absence of side-effects
- most prominently: Haskell
- useful for operating on *infinite* data structures (for instance: streams)

8.6.14 Lazy evaluation / streams

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

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

8.7 Virtual methods in OO

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

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

We earlier also mentioned *dynamic dispatch*. Dispatch is what the run-time system has to do when calling a procedure, function, method etc., i.e., *jumping* to the beinning of the code of body of the procedure (plus performing the steps of the call sequence). Determining the jump target, i.e., locating the code of the procedure, can be done at compile time or a run-time. That’s *static dispatch* resp. *dynamic dispatch*.

Preferable, efficiency-wise, is static dispatch. If the compiler can determine the jump-target, then it can produce code with the jump address “hard-coded” into the jump-command. When writing, for intstance, `x.m()`, establishing the connection between the source-code level name `m` of the method and the corresponding method body, ultimately the address in the code section, that’s also called *binding*. Binding names to addresses is, of course, more general than for methods or procedure names only. The associatiob of `x` with its address is likewise called `binding`. We have mostly looked a statically bound variables (also called lexically bound). For instance, the use of static links to locate the proper address of a statically bound variable in languages with nested procedures.

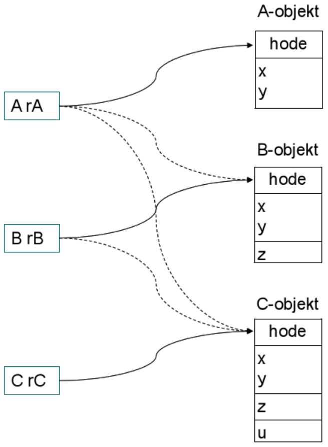
Static binding is impossible for late-bound methods, resp. late-bound just means the binding is done at run-time, i.e., dynamically not statically. The concept of late bound method is central for the concept of [. . . continue here. . .] The need for dynamic binding for virtual methods is ultimately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

8.7.1 Object-orientation

- class-based/inheritance-based OO
- classes and sub-classes
- typed references to objects
- *virtual* and *non-virtual* methods

8.7.2 Virtual and non-virtual methods + fields

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

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

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



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

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

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

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

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

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

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

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

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

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

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

8.7.3 Call to virtual and non-virtual methods

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

non-virtual method f

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

virtual methods g and h

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

8.7.4 Late binding/dynamic binding

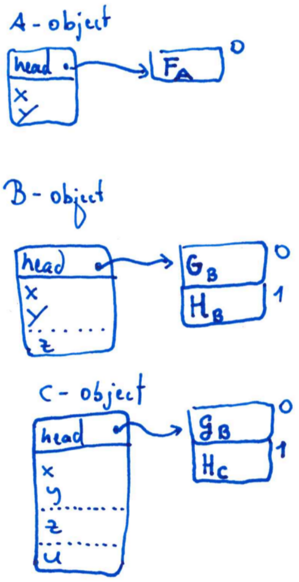
- details very much depend on the language/flavor of OO
  - single vs. multiple inheritance?
  - method update, method extension possible?
  - how much information available (e.g., static type information)?
- simple approach: "embedding" methods (as references)
  - seldomly done (but needed for updateable methods)
- using *inheritance graph*
  - each object keeps a pointer to its class (to locate virtual methods)
- virtual function table
  - in static memory
  - no traversal necessary
  - class structure need be known at compile-time
  - C^{++}

8.7.5 Virtual function table

- static check ("type check") of $r_X.f()$
  - for virtual methods: `f` must be defined in X or one of its superclasses
- non-virtual binding: finalized by the compiler (static binding)
- virtual methods: enumerated (with offset) from the first class with a virtual method, redefinitions get the same "number"
- object "headers": point to the class's **virtual function table**
- $r_A.g()$:

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

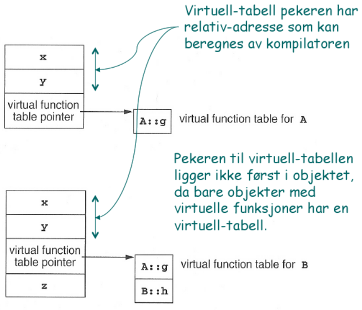
- compiler knows
  - `g_offset = 0`
  - `h_offset = 1`



8.7.6 Virtual method implementation in C++

- according to [?]

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

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


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

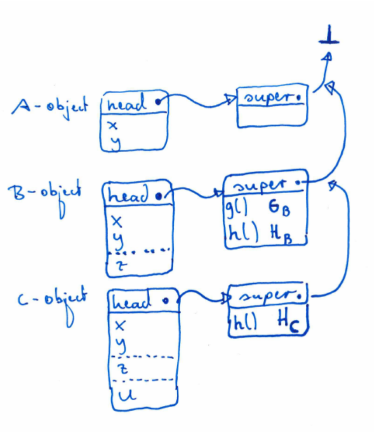
Virtuell-tabell pekeren har relativ-adresse som kan beregnes av kompilatoren

Pekeren til virtuell-tabellen ligger ikke først i objektet, da bare objekter med virtuelle funksjoner har en virtuell-tabell.

8.7.7 Untyped references to objects (e.g. Smalltalk)

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

- *search* for g following the superclass hierarchy.



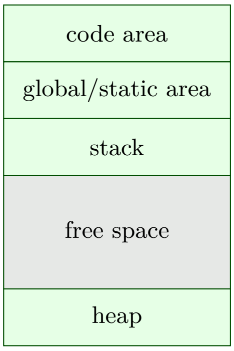
8.8 Garbage collection

8.8.1 Management of dynamic memory: GC & alternatives

- *dynamic* memory: allocation & deallocation at *run-time*
- different alternatives
  1. manual
     - "alloc", "free"
     - error prone
  2. "stack" allocated dynamic memory
     - typically not called GC
  3. automatic *reclaim* of unused dynamic memory
     - requires extra provisions by the compiler/RTE

8.8.2 Heap

- "heap" unrelated to the well-known heap-data structure from A&D
- part of the *dynamic* memory
- contains typically
  - objects, records (which are dynamocally allocated)
  - often: arrays as well
  - for "expressive" languages: heap-allocated activation records
    * coroutines (e.g. Simula)
    * higher-order functions



Memory

8.8.3 Problems with free use of pointers

```
  int * dangle (void) {
    int x;       // local var
    return &x;   // address of x
  }
```

```
typedef int (* proc) (void);

proc g(int x) {
  int f(void) { /* illegal  */
    return x;
  }
  return f;
}

main () {
  proc c;
  c = g(2);
  printf("%d\n", c()); /* 2? */
  return 0;
}
```

- as seen before: references, higher-order functions, coroutines etc ⇒ heap-allocated ARs
- higher-order functions: typical for functional languages,
- heap memory: no LIFO discipline
- *unreasonable* to expect user to "clean up" AR's (already `alloc` and `free` is error-prone)
- ⇒ garbage collection (already dating back to 1958/Lisp)

8.8.4 Some basic design decisions

- gc *approximative*, but non-negotiable condition: **never** reclaim cells which *may* be used in the future
- one basic decision:
  1. never *move* "objects"
     - may lead to fragmentation
  2. *move* objects which are still needed
     - extra administration/information needed
     - all reference of moved objects need adaptation
     - all free spaces collected adjacently (defragmentation)

- *when* to do gc?
- *how* to get info about definitely unused/potentially used obects?
  - "monitor" the interaction program ↔ heap while it *runs*, to keep "up-to-date" all the time
  - inspect (at approriate points in time) the *state* of the heap

Objects here are meant as heap-allocated entities, which in OO languages includes objects, but here referring also to other data (records, arrays, closures . . .).

8.8.5 Mark (and sweep): marking phase

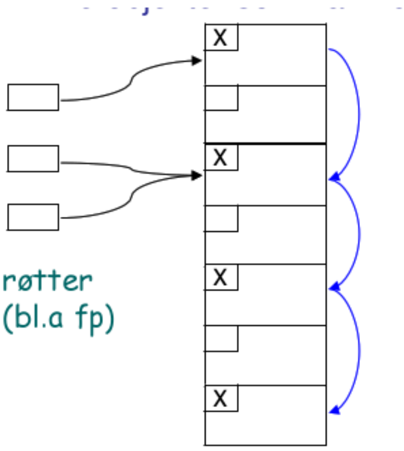
- observation: heap addresses only **reachable**

  directly through variables (with references), kept in the run-time stack (or registers)

  indirectly following fields in reachable objects, which point to further objects . . .

- heap: *graph* of objects, entry points aka "roots" or *root set*
- *mark*: starting from the root set:
  - find reachable objects, *mark* them as (potentially) used
  - one boolean (= 1 *bit* info) as mark
  - depth-first search of the graph

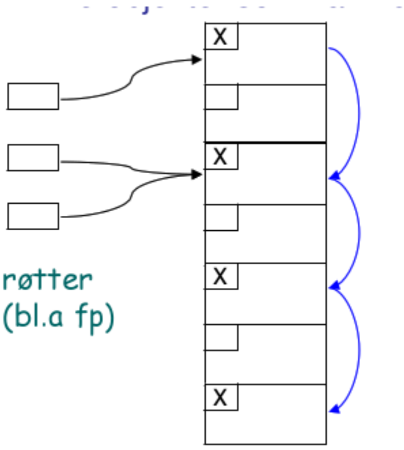
8.8.6 Marking phase: follow the pointers via DFS



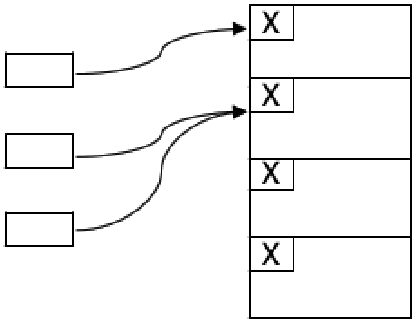
- layout (or "type") of objects need to be known to determine where pointers are
- food for thought: doing DFS requires a *stack*, in the worst case of comparable size as the heap itself . . . .

8.8.7 Compactation

Marked



Compacted

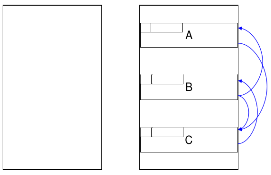


8.8.8 After marking?

- known *classification* in “garbage” and “non-garbage”
- pool of “unmarked” objects
- however: the “free space” not really ready at hand:
- two options:
  1. *sweep*
     - go again through the heap, this time sequentially (no graph-search)
     - collect all unmarked objects in **free list**
     - objects remain at their place
     - RTE need to allocate new object: grab free slot from free list
  2. *compaction* as well:
     - avoid fragmentation
     - move non-garbage to one place, the rest is big free space
     - when *moving* objects: adjust pointers

8.8.9 Stop-and-copy

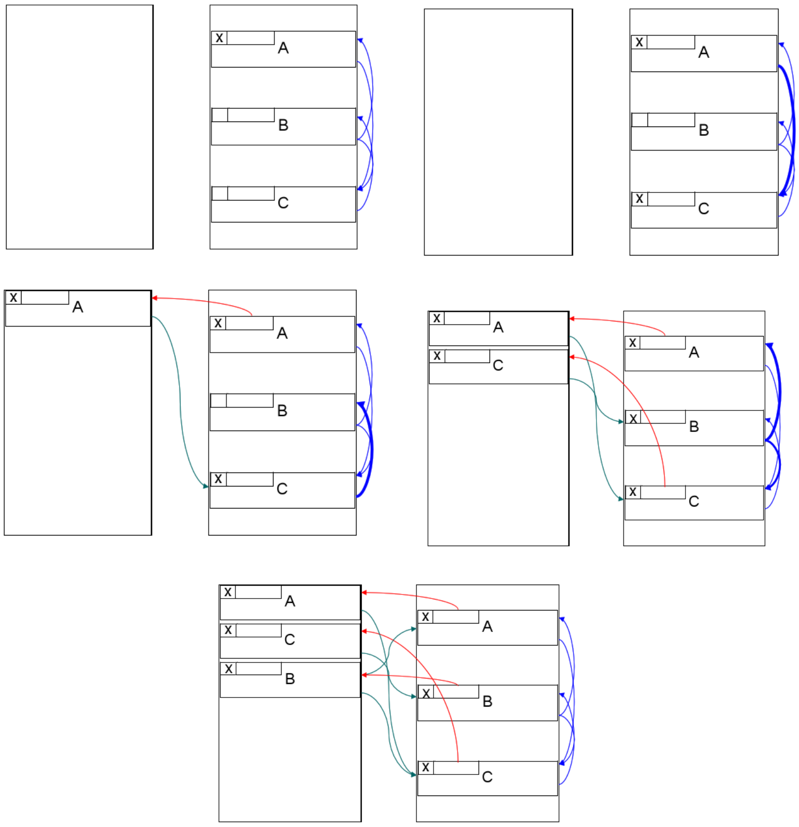
- variation of the previous compactation
- mark & compactation can be done in recursive pass
- space for heap-managment
  - split into *two halves*
  - only one half used at any given point in time
  - compactation by copying all non-garbage (marked) to the currently unused half



Hvert objekt må
ha et ledig bit
(”er flyttet”)

Da angir ”neste
ordet” adressen
det er flyttet til

8.8.10 Step by step



Index