

Chapter 8

Run-time environments

Course “Compiler Construction”

Martin Steffen

Spring 2022



Chapter 8

Learning Targets of Chapter "Run-time environments".

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



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection



Chapter 8

Outline of Chapter “Run-time environments”.

Intro

Different layouts

- Full static layout
- Stack-based runtime environments
- Stack-based RTE with nested procedures
- Functions as parameters

Parameter passing

Virtual methods in OO

Garbage collection

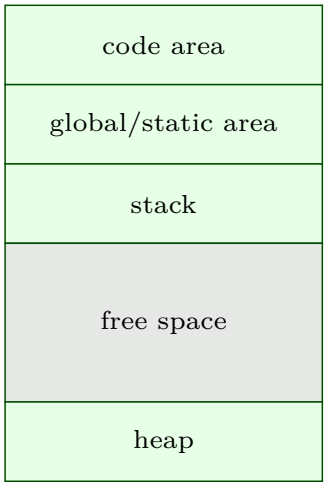


Section

Intro

Chapter 8 “Run-time environments”
Course “Compiler Construction”
Martin Steffen
Spring 2022

Static & dynamic memory layout at runtime

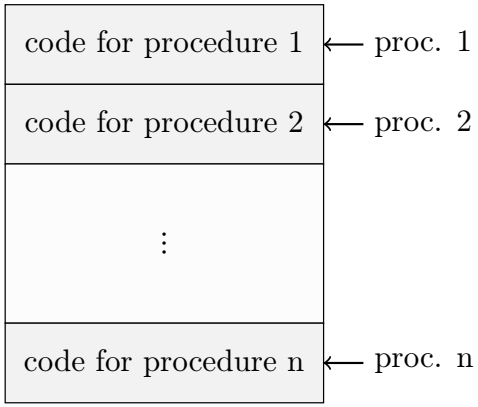


Memory

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

- static memory
- dynamic memory:
  - stack
  - heap

Translated program code



Code memory

- code segment: almost always considered as statically allocated

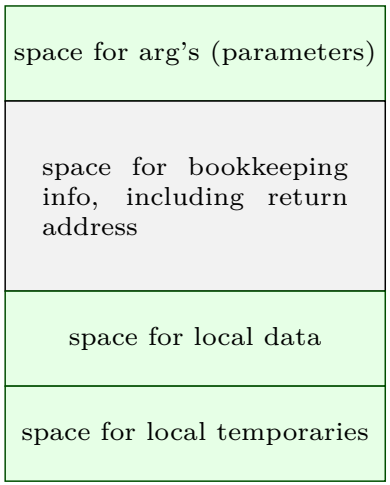
$\Rightarrow$ neither moved nor changed at runtime

- 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*



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Activation records



- *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





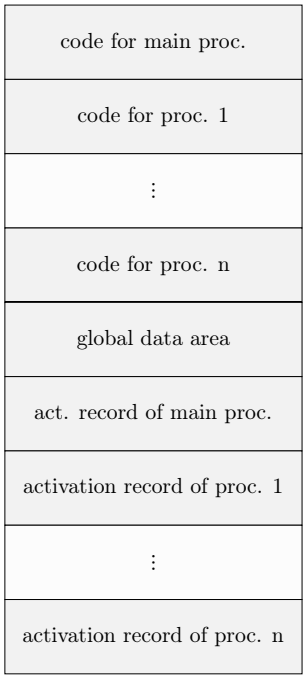
Section

Different layouts

- Full static layout
- Stack-based runtime environments
- Stack-based RTE with nested procedures
- Functions as parameters

Chapter 8 “Run-time environments”
Course “Compiler Construction”
Martin Steffen
Spring 2022

Full static layout



- static addresses of all of the memory known to the compiler
  - executable code
  - variables
  - all forms of auxiliary data (for instance big constants in the program, e.g., string literals)
- for instance: (old) Fortran
- nowadays rather seldom (or special applications like safety critical embedded systems)



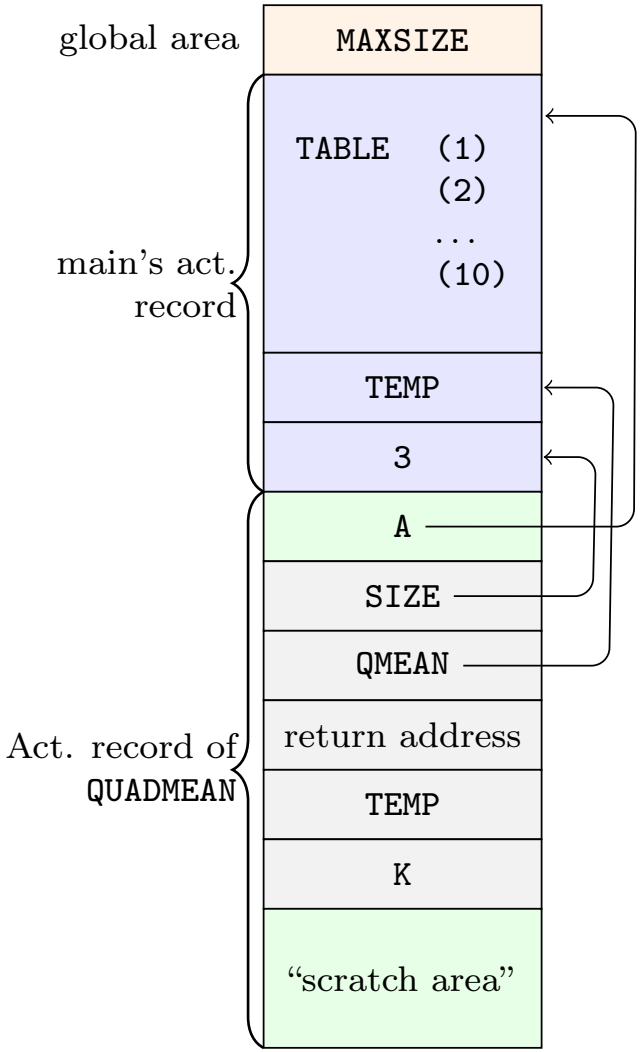
Fortran example

```
1         PROGRAM TEST
2         COMMON MAXSIZE
3         INTEGER MAXSIZE
4         REAL TABLE(10),TEMP
5         MAXSIZE = 10
6         READ *, TABLE(1),TABLE(2),TABLE(3)
7         CALL QUADMEAN(TABLE,3,TEMP)
8         PRINT *,TEMP
9         END
10
11        SUBROUTINE QUADMEAN(A,SIZE,QMEAN)
12        COMMON MAXSIZE
13        INTEGERMAXSIZE,SIZE
14        REAL A(SIZE),QMEAN, TEMP
15        INTEGER K
16        TEMP = 0.0
17        IF ((SIZE.GT.MAXSIZE).OR.(SIZE.LT.1)) GOTO 99
18        DO 10 K = 1, SIZE
19            TEMP = TEMP + A(K)*A(K)
20    10  CONTINUE
21    99  QMEAN = SQRT(TEMP/SIZE)
22        RETURN
23        END
```



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Static memory layout example/runtime environment



Static memory layout example/runtime environment



in Fortan (here Fortran77)

- parameter passing as *pointers* to the actual parameters
- activation record for `QUADMEAN` contains place for intermediate results, compiler calculates, how much is needed.
- note: one possible memory layout for FORTRAN 77, details vary, other implementations exists as do more modern versions of Fortran

Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

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*

$\Rightarrow$ 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



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

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*

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



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

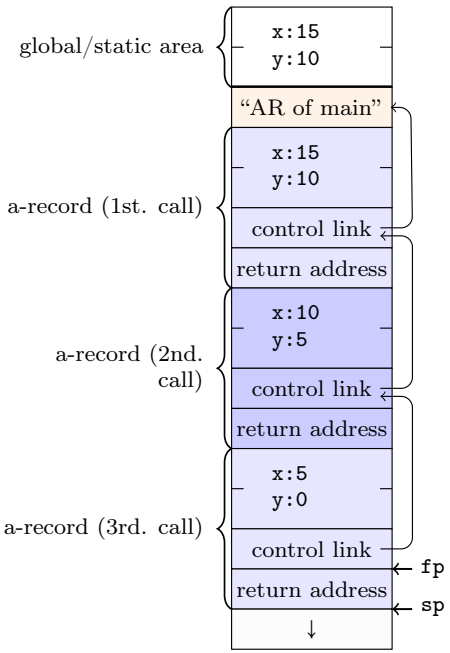
Euclid's recursive gcd algo

```
1  #include <stdio.h>
2
3  int x,y;
4
5  int gcd (int u, int v)
6  { if (v==0) return u;
7       else return gcd(v,u % v);
8  }
9
10 int main ()
11 { scanf("%d%d",&x,&y);
12    printf("%d\n",gcd(x,y));
13    return 0;
14 }
```



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

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



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Local and global variables and scoping

```
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



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Activation records and activation trees



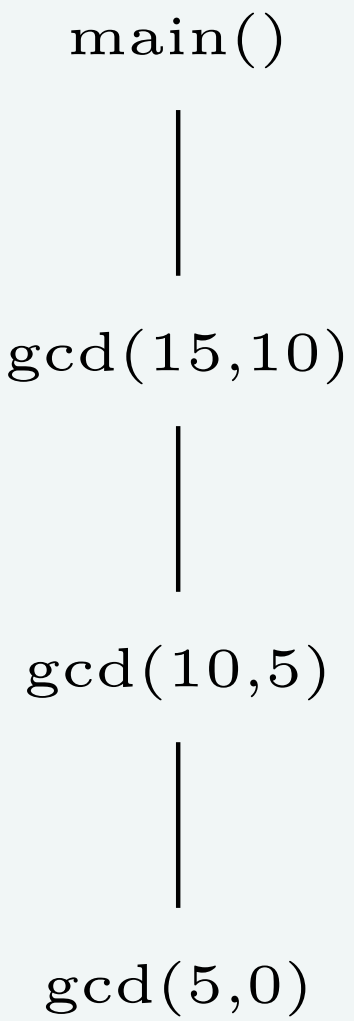
- *activation* of a function: corresponds to: *call* of a function
- activation record
  - data structure for run-time system
  - holds all relevant data for a function call and control-info in "standardized" form
  - control-behavior of functions: LIFO
  - if data *cannot* outlive activation of a function

  $\Rightarrow$ activation records can be arranged in as stack (like here)
  - in this case: activation record AKA *stack frame*

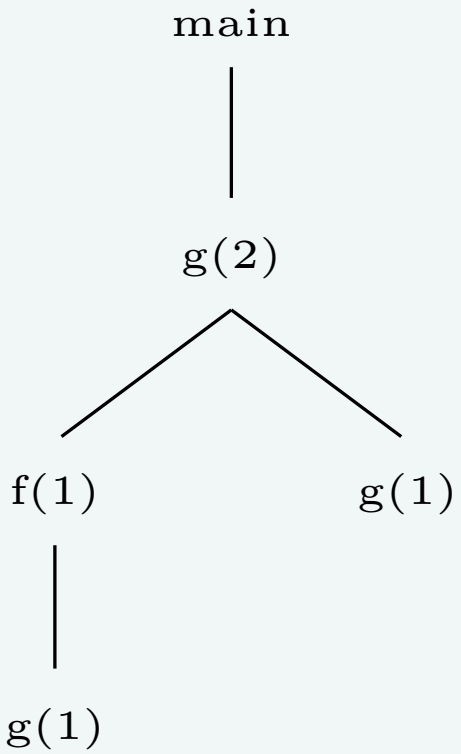
Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Activation record and activation trees

GCD



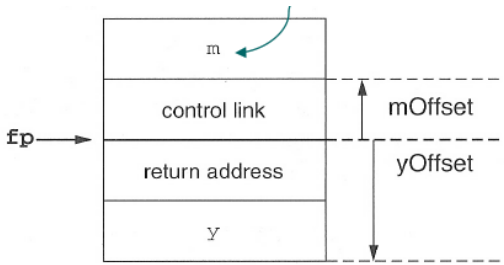
f and g example





Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Variable access and design of ARs



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

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

$\Rightarrow$ *frames* on the stack differently sized

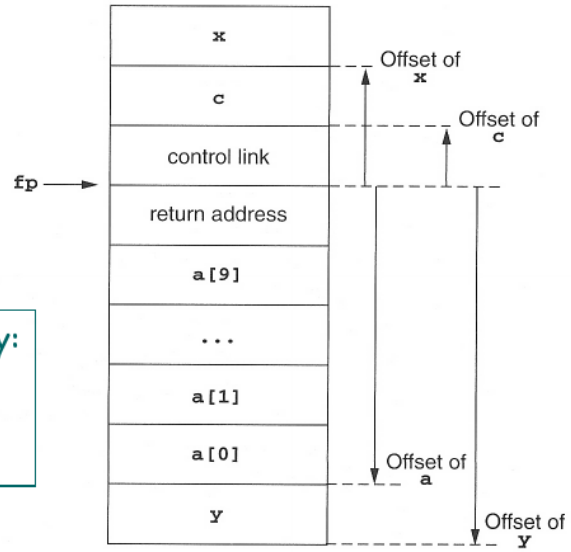
- note: FP points
  - not: "top" of the frame/stack, but
  - to a well-chosen, well-defined position in the frame
  - other local data (local vars) accessible *relative* to that
- conventions
  - higher addresses "higher up"
  - stack "grows" towards lower addresses



Layout for arrays of statically known size

```
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)
```

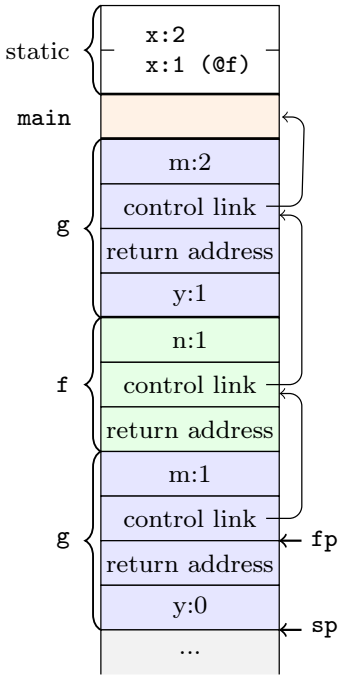
Back to the C code again (global and local variables)

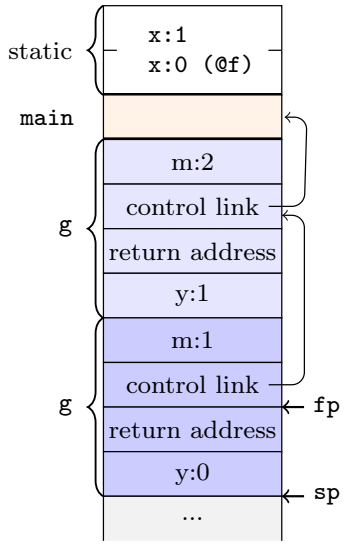
```
1  int x  = 2; /* glob. var */
2  void g(int);/* prototype */
3
4  void f(int n)
5    { static int x = 1;
6      g(n);
7      x--;
8    }
9
10 void g(int m)
11   { int y = m-1;
12     if (y > 0)
13       { f(y);
14         x--;
15        g(y);
16       }
17   }
18
19 int  main ()
20   { g(x);
21     return 0;
22   }
```



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

2 snapshots of the call stack





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



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

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
  - ...



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

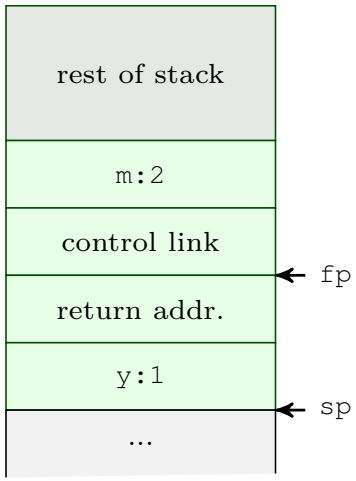
Steps when calling

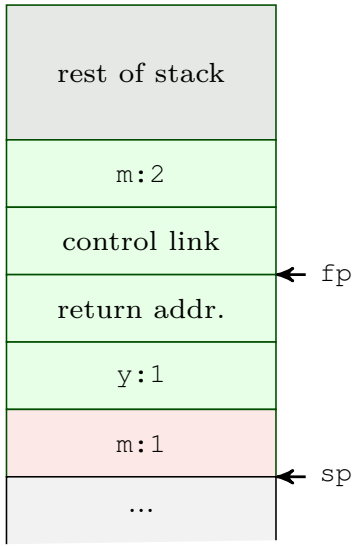
- For procedure call (entry)
  1. compute arguments, store them in the correct positions in the *new* activation record of the procedure (pushing them in order onto the runtime stack will achieve this)
  2. store (push) the `fp` as the *control link* in the new activation record
  3. change the `fp`,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

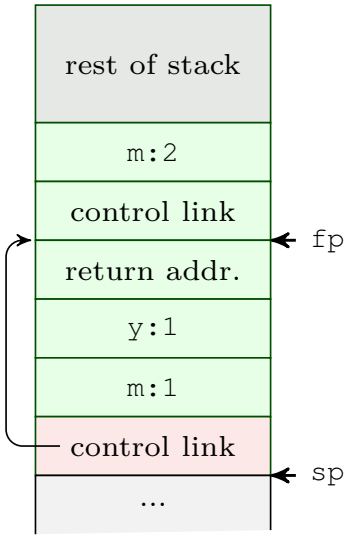


Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

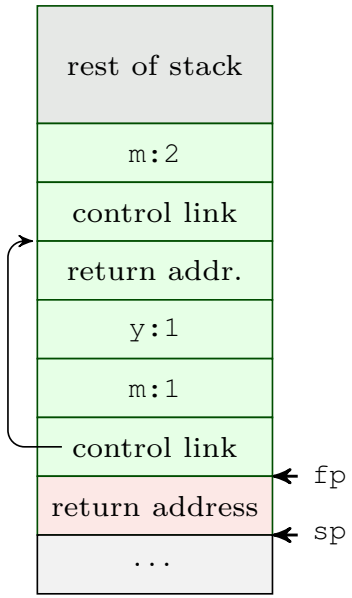
Steps when calling g

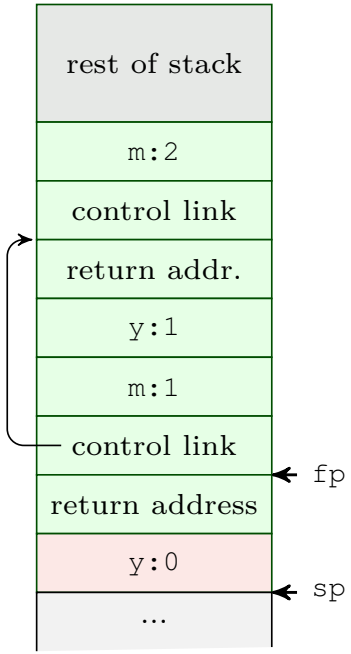




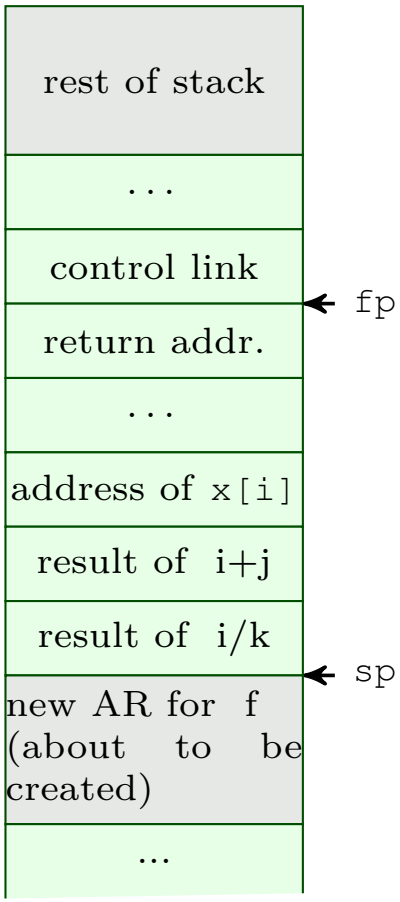


Steps when calling g (cont'd)





Treatment of auxiliary results: "temporaries"



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

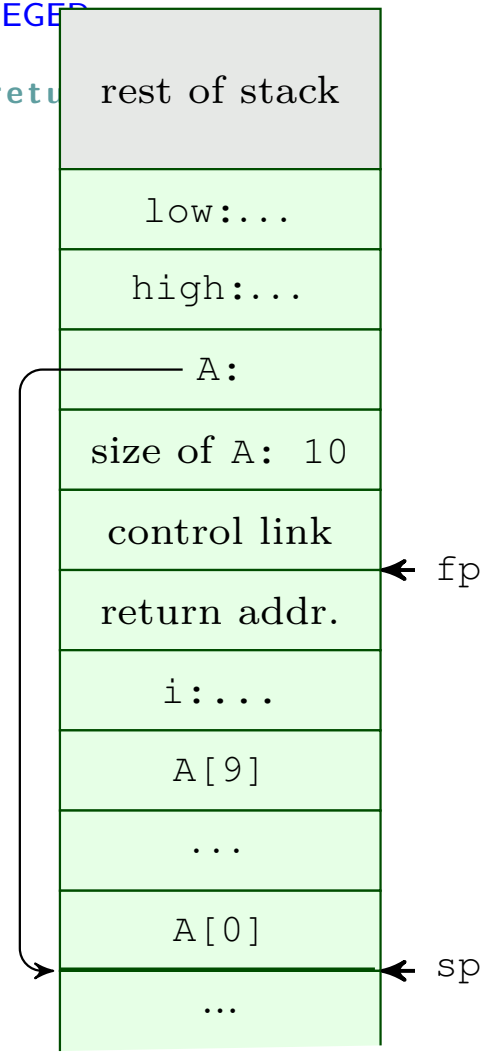
```
x[i] = (i + j) * (i/k + f(j));
```

- 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

Variable-length data

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

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

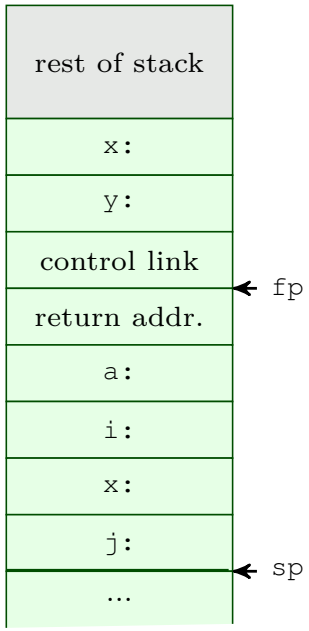


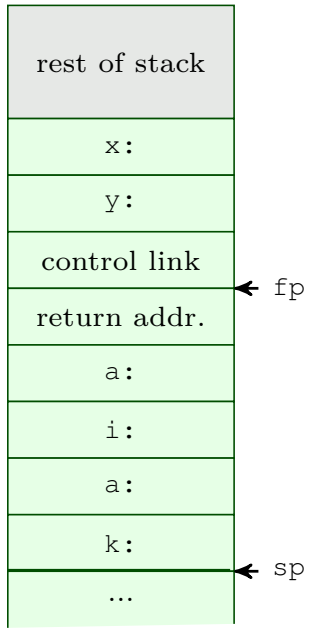
- Ada example
- assume: array passed *by value* ("copying")
- `A[i]`: calculated as `@6(fp) + 2*i`
- in Java and other languages: arrays passed *by reference*
- note: space for `A` (as ref) and size of `A` is fixed-size (as well as `low` and `high`)



Nested declarations ("compound statements")

```
1  void p(int x, double y)
2  { char a;
3    int i;
4    ...;
5   A:{ double x;
6      int j;
7      ...;
8    }
9    ...;
10  B: { char * a;
11     int k;
12     ...;
13   };
14   ...;
15 }
```





Nested procedures in Pascal

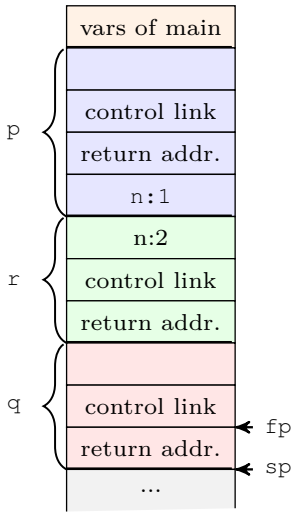
```
1  program nonLocalRef;
2  procedure p;
3      var n : integer;
4      procedure q;
5      begin
6          (* a ref to n is now non-local, non-global *)
7      end; (* q *)
8
9      procedure r(n : integer);
10     begin
11         q;
12     end; (* r *)
13
14 begin (* p *)
15     n := 1;
16     r(2);
17 end; (* p *)
18
19 begin (* main *)
20     p;
21 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`



Accessing non-local var's

- `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

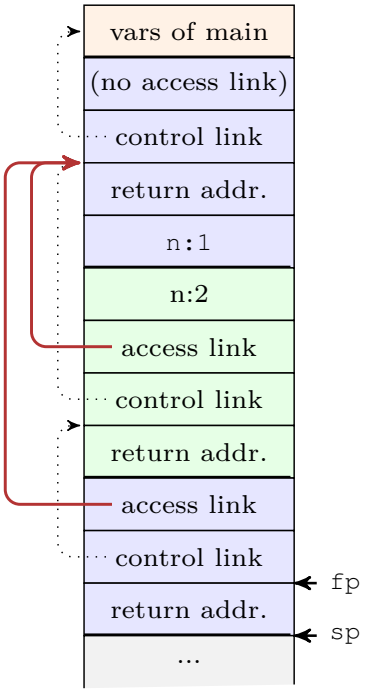
- "name-addressable" mapping
- access at compile time
- cf. scope tree

Dynamic memory

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



Access link as part of the AR



- 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

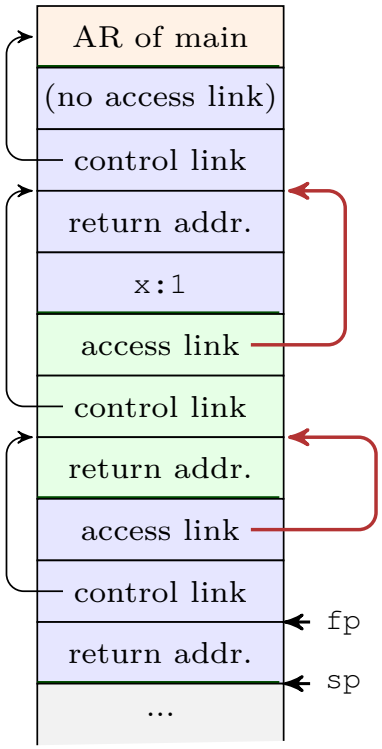
Example with multiple levels

```
1  program chain;
2
3  procedure p;
4  var x : integer;
5
6     procedure q;
7        procedure r;
8        begin
9           x:=2;
10          ...;
11          if ... then p;
12       end; (*r *)
13    begin
14       r;
15    end; (* q *)
16
17 begin
18    q;
19 end; (* p *)
20
21 begin (* main *)
22    p;
23 end.
```



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Access chaining



- 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



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



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

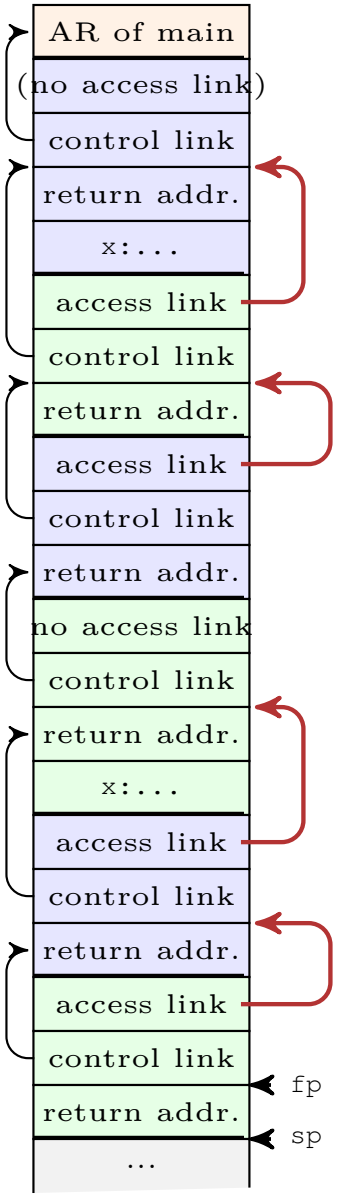
Calling sequence

- 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



Calling sequence: with access links



- `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



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Another frame design (Tiger)

lstinputlisting[language=ocaml] /cor/tiger/src/compiler/frames.ml

- 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[2]
- often: stack based allocation + fully-dynamic (= heap-based) allocation

[2]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.



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

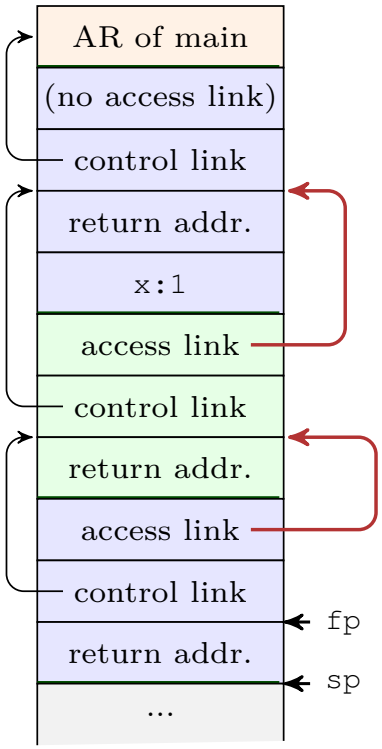
Example with multiple levels

```
1  program chain;
2
3  procedure p;
4  var x : integer;
5
6     procedure q;
7        procedure r;
8        begin
9           x:=2;
10          ...;
11          if ... then p;
12       end; (*r *)
13    begin
14       r;
15    end; (* q *)
16
17 begin
18    q;
19 end; (* p *)
20
21 begin (* main *)
22    p;
23 end.
```



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Access chaining



- 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



Procedures as parameters

```
1  program closureex(output);
2
3  procedure p(procedure a);
4  begin
5     a;
6  end;
7
8  procedure q;
9  var x : integer;
10    procedure r;
11    begin
12       writeln(x);    // ``non-local''
13    end;
14
15 begin
16    x := 2;
17    p (r);
18 end; (* q *)
19
20 begin (* main *)
21    q;
22 end.
```



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Procedures as parameters, same example in Go

```
1  package main
2  import ("fmt")
3
4  var p = func (a (func () ())) {  // (unit -> unit) -> unit
5          a()
6  }
7
8  var q = func () {
9          var x = 0
10         var r = func () {
11         fmt.Printf(" x = %v", x)
12         }
13         x = 2
14         p(r)     // r as argument
15 }
16
17
18 func main() {
19         q();
20 }
```



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Procedures as parameters, same example in ocaml

```
1  let p (a :unit -> unit) : unit =   a();;
2
3  let q() =
4    let x: int ref  =  ref 1
5    in let r = function () ->  (print_int !x) (* deref *)
6    in
7    x := 2;     (* assignment to ref-typed var *)
8    p(r);;
9
10 q();;  (* ``body of main'' *)
```



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

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*



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

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[3] *together* with the values for all its variables, including the non-local ones.[3]

- 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)

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



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

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?

data problem

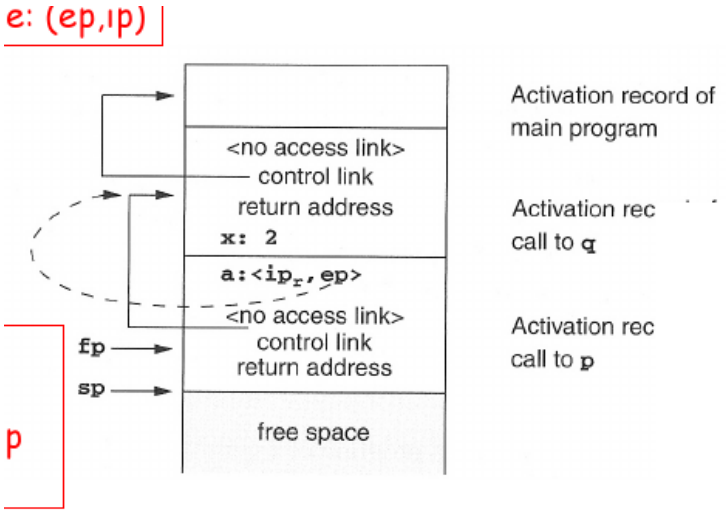
How can one statically arrange that `a` will be able to access non-local variables if statically it's not known what `a` will be?

- 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!



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

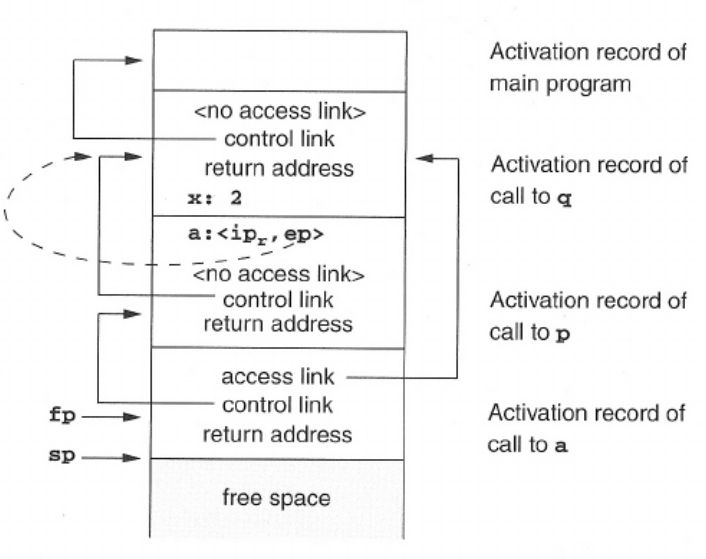
Closure for formal parameter a of the example



- 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)



After calling a (= r)

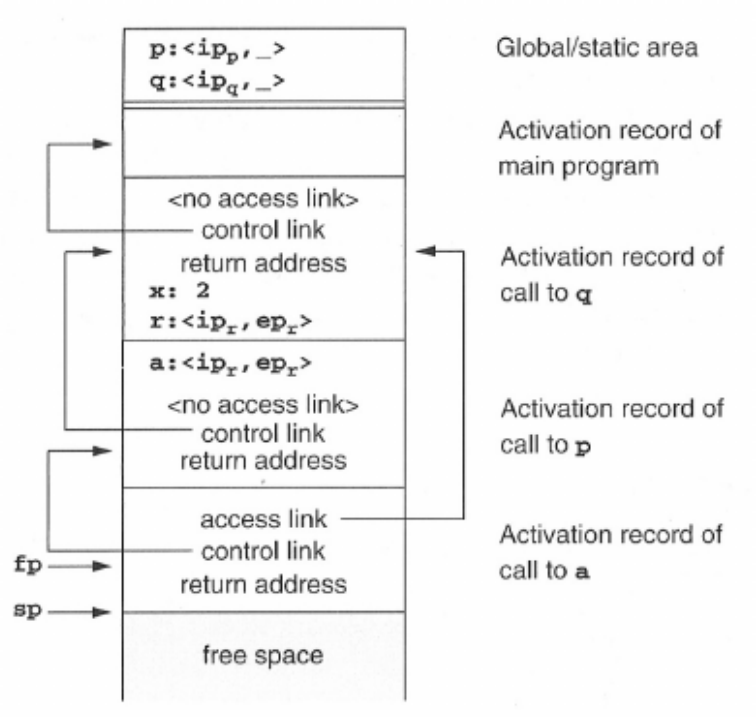


- note: *static* link of the new frame: used from the closure!



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Making it uniform



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



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.



Dangling ref's due to returning references

```
1  int * dangle (void) { q// return type: pointer to an int
2      int x;            // local var
3      return &x;        // address of x
4    }
```

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



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Function variables

NCLUDE: "../code/typechecking/funcvar.pas" src pascal

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



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Functions as return values

```
1  package main
2  import ("fmt")
3
4  var f = func () (func (int) int)  { // unit -> (int -> int)
5          var x = 40                   // local variable
6          var g = func (y int) int { // nested function
7                  return x + 1
8          }
9          x = x+1                      // update x
10         return g                     // function as return value
11 }
12
13 func main() {
14         var x = 0
15         var h = f()
16         fmt.Println(x)
17         var r = h (1)
18         fmt.Printf(" r = %v", r)
19 }
```

INCLUDE: "~/cor/teaching/compilerconstruction/src/s-lides/code/runtimeenv/" src go

- function `g`
  - defined local to `f`



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

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



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection



Section

Parameter passing

Chapter 8 “Run-time environments”
Course “Compiler Construction”
Martin Steffen
Spring 2022

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



CBV and CBR, roughly

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]



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Parameter passing by-value

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

```
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



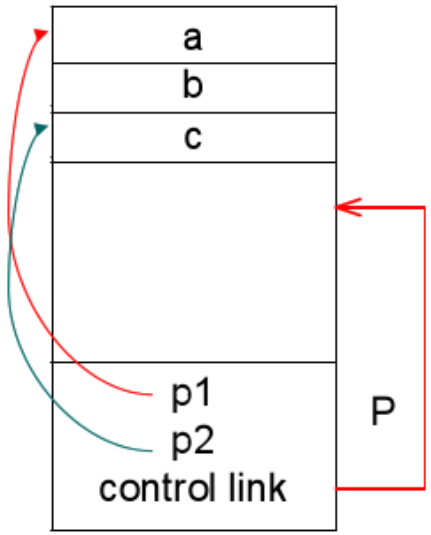
Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

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

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



Listing: Pointers as arguments

Call-by-value, call-by-reference, or what?

```
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
    }
};
```

Guess (and try out), what's printed? The explanation is not (just) connected with parameter passing.



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

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 $\rightarrow$ formal parameters
  - when returning: actual $\leftarrow$ 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 $\leftarrow$ formal parameters"
  - when calling
  - when returning
- not the cleanest parameter passing mechanism around...



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Call-by-value-result example

```
void p(int x, int y)
{
  ++x;
  ++y;++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"[4]

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



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)



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

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

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

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

```
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;
}
```



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



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Call-by-name illustrations

```
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



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

  $\Rightarrow$ 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)



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

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)
```



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection



Section

Virtual methods in OO

Chapter 8 “Run-time environments”
Course “Compiler Construction”
Martin Steffen
Spring 2022

Object-orientation

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



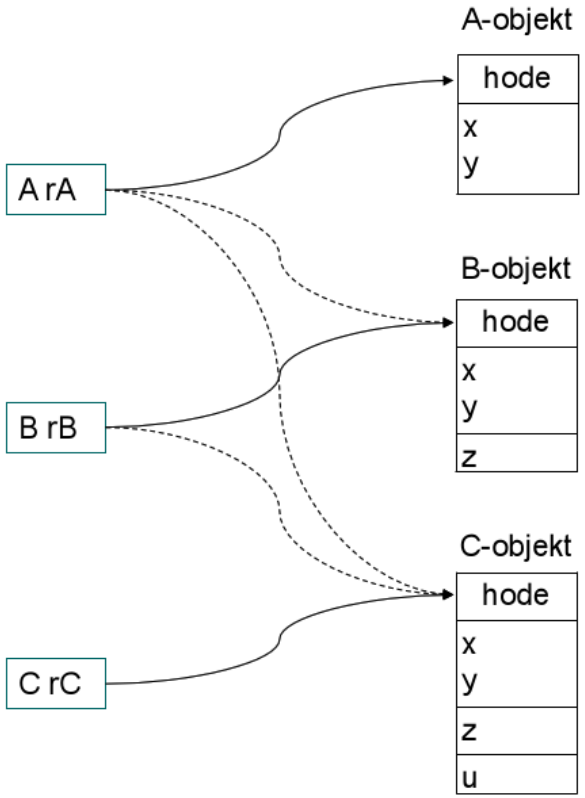
Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

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 ... };
}
```



Listing: Virtual and non-virtual methods + fields

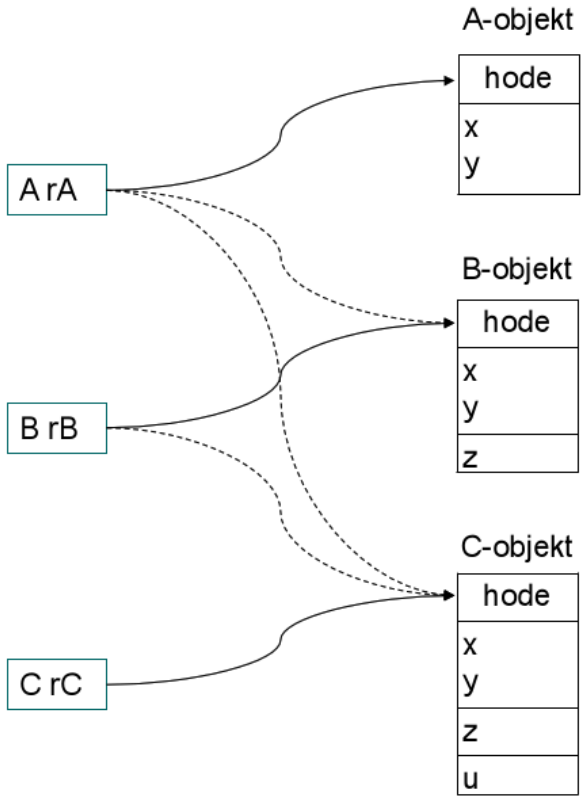
Call to virtual and non-virtual methods

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



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++

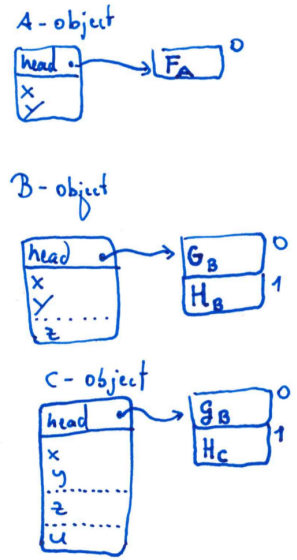


Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

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
- $r_A.g()$:

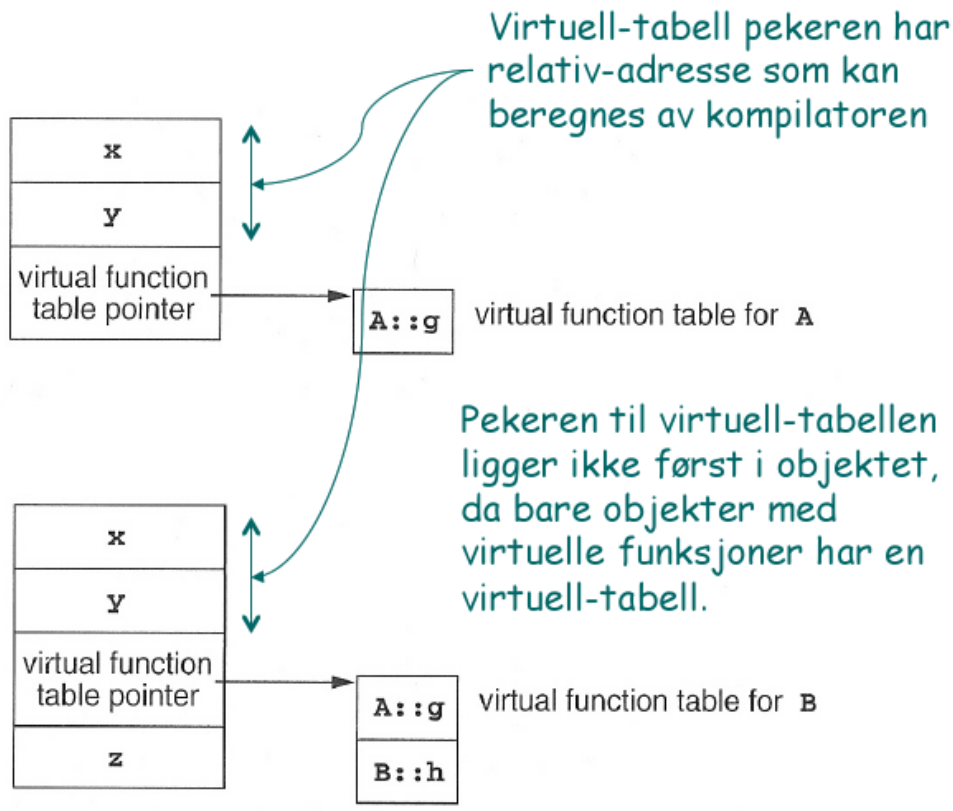
```
call r_A.virttab[g_offset]
```



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

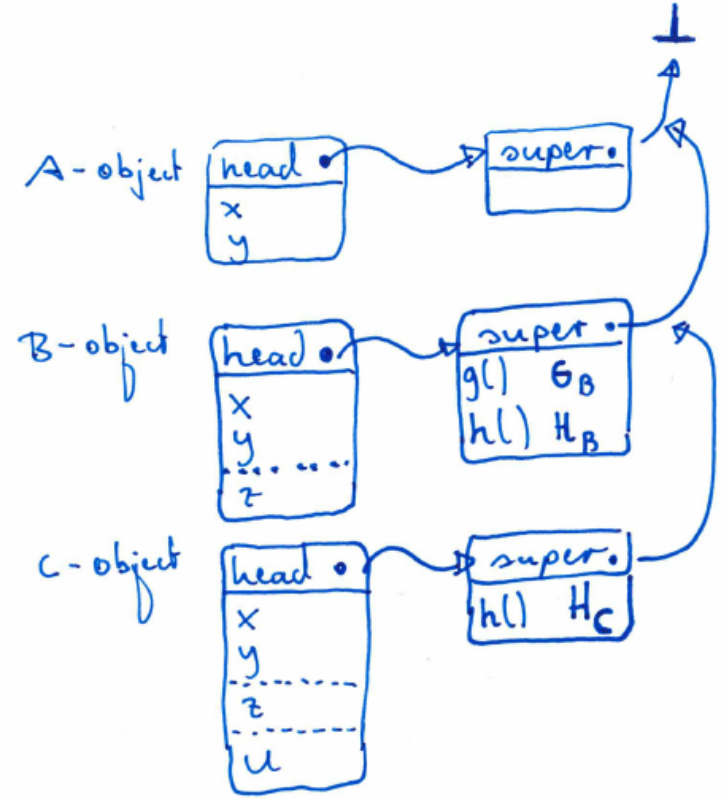
Virtual method implementation in C++

- according to [**?**]



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.





Section

Garbage collection

Chapter 8 “Run-time environments”
Course “Compiler Construction”
Martin Steffen
Spring 2022

Management of dynamic memory: GC & alternatives

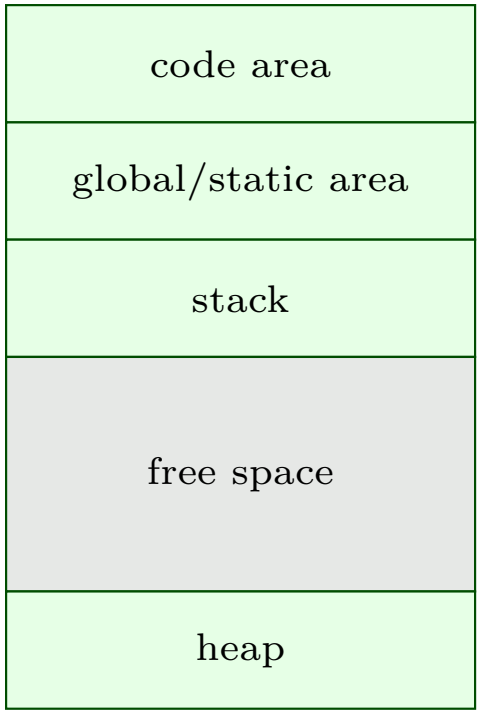


- *dynamic* memory: allocation & deallocation at *run-time*
- different alternatives
  1. manual
     - "alloc", "free"
     - error prone
  2. "stack" allocated dynamic memory
     - typically not called GC
  3. automatic *reclaim* of unused dynamic memory
     - requires extra provisions by the compiler/RTE

Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

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

Problems with free use of pointers

```
int * dangle (void) { q// return type: pointer to an int
    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)



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 $\leftrightarrow$ heap while it *runs*, to keep "up-to-date" all the time
  - inspect (at approriate points in time) the *state* of the heap



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

Mark (and sweep): marking phase

- observation: heap addresses only reachable

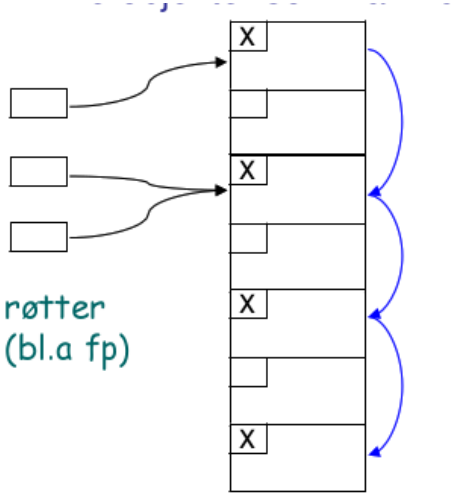
  directly through variables (with references), kept in the run-time stack (or registers)

  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



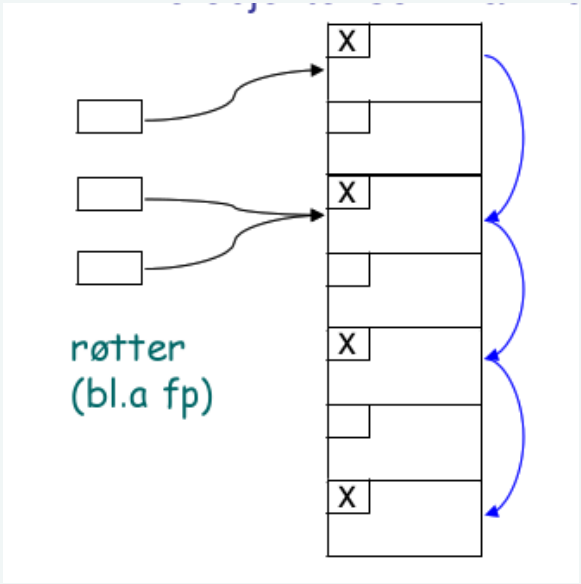
Marking phase: follow the pointers via DFS



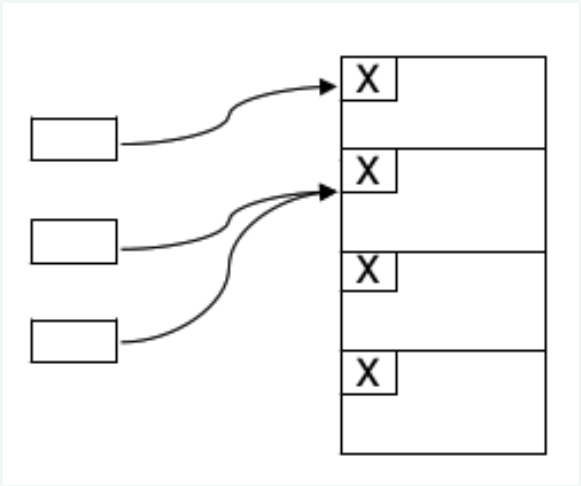
- layout (or "type") of objects need to be known to determine where pointers are
- food for thought: doing DFS requires a *stack*, in the worst case of comparable size as the heap itself ....

Compactation

Marked



Compacted



Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

After marking?

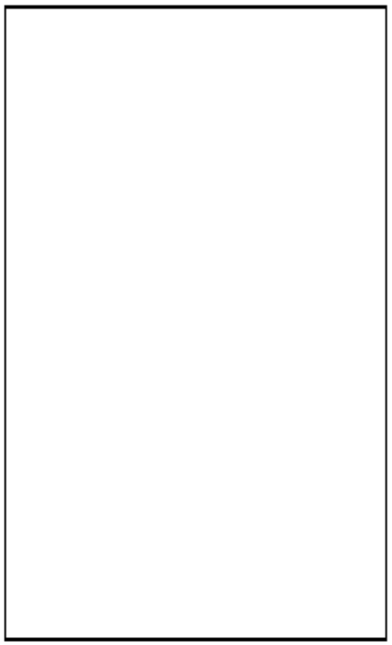
- known *classification* in "garbage" and "non-garbage"
- pool of "unmarked" objects
- however: the "free space" not really ready at hand:
- two options:
  1. *sweep*
     - go again through the heap, this time sequentially (no graph-search)
     - collect all unmarked objects in free list
     - objects 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

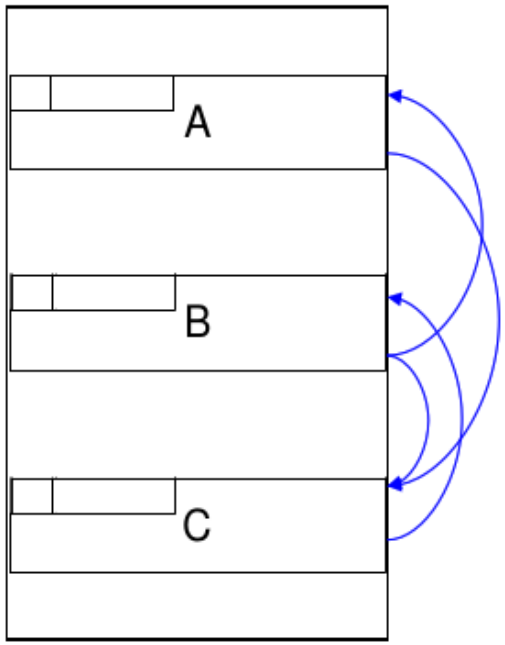


Targets & Outline
Intro
Different layouts
Full static layout
Stack-based runtime environments
Stack-based RTE with nested procedures
Functions as parameters
Parameter passing
Virtual methods in OO
Garbage collection

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



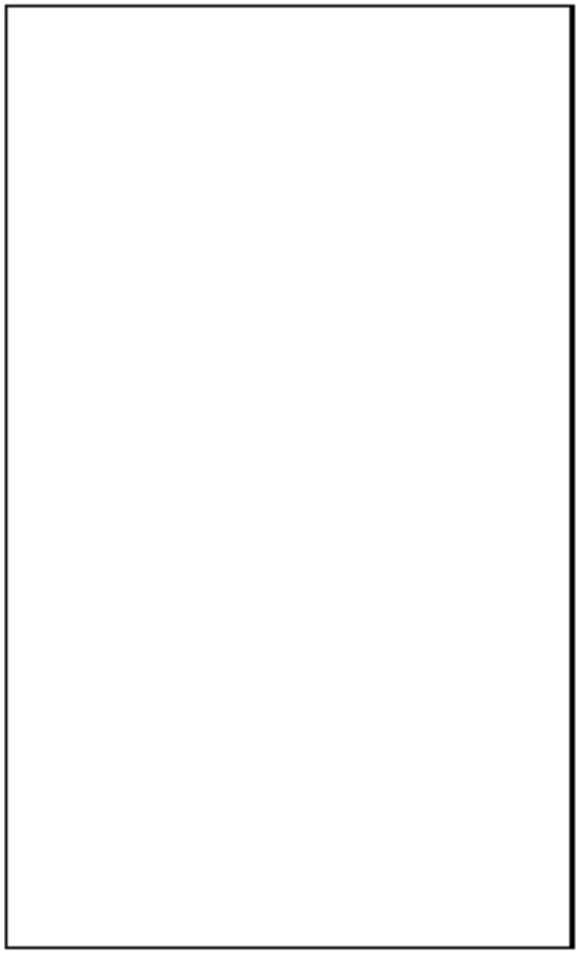


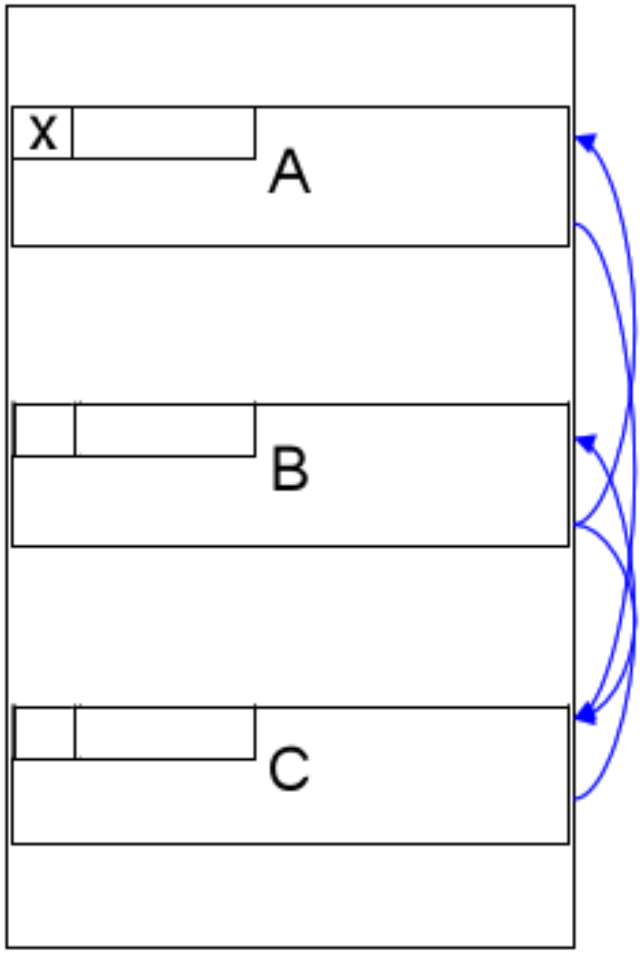
Hvert objekt må ha et ledig bit ("er flyttet")

Da angir "neste ordet" adressen det er flyttet til

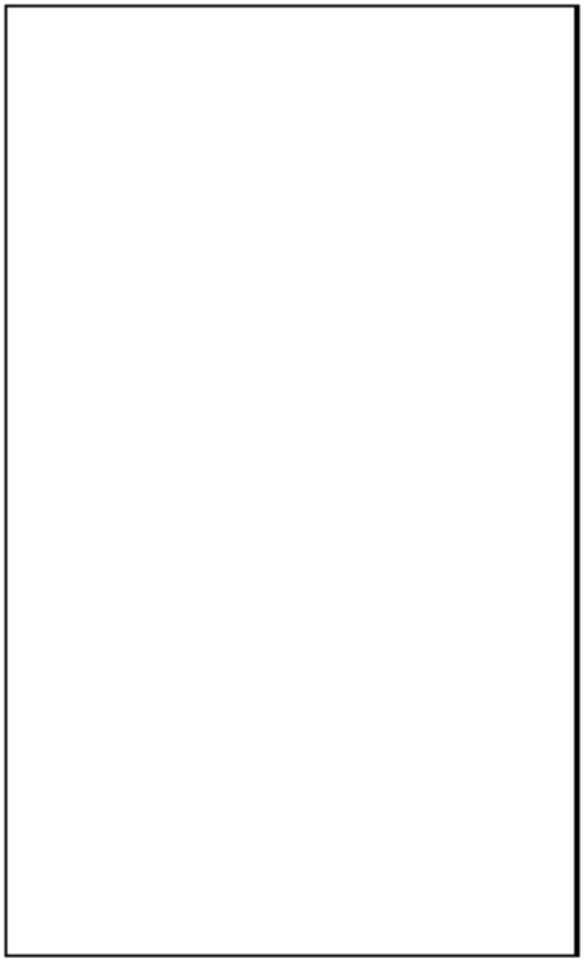


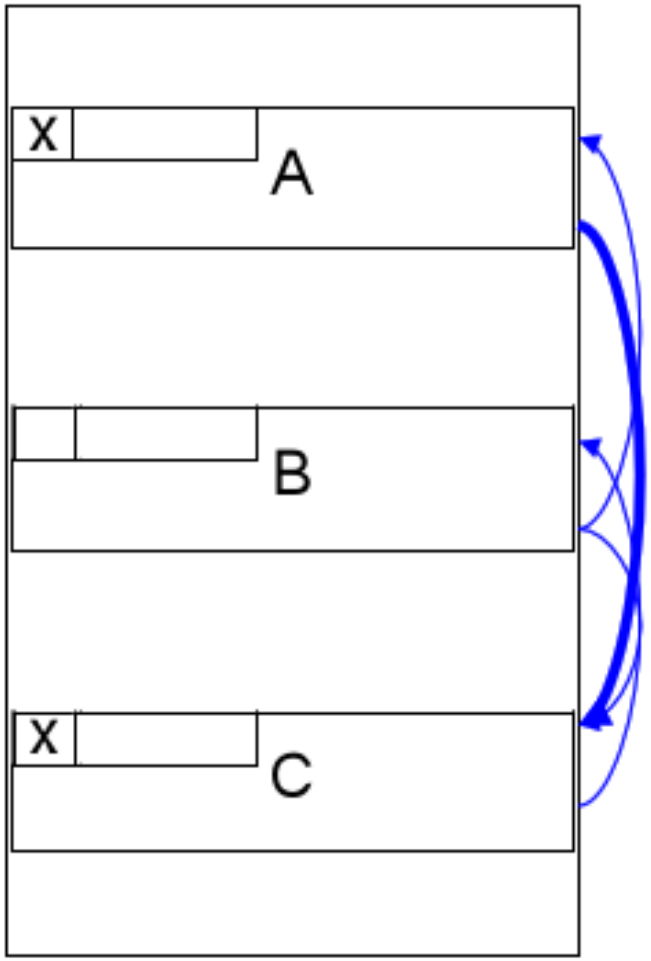
Step by step



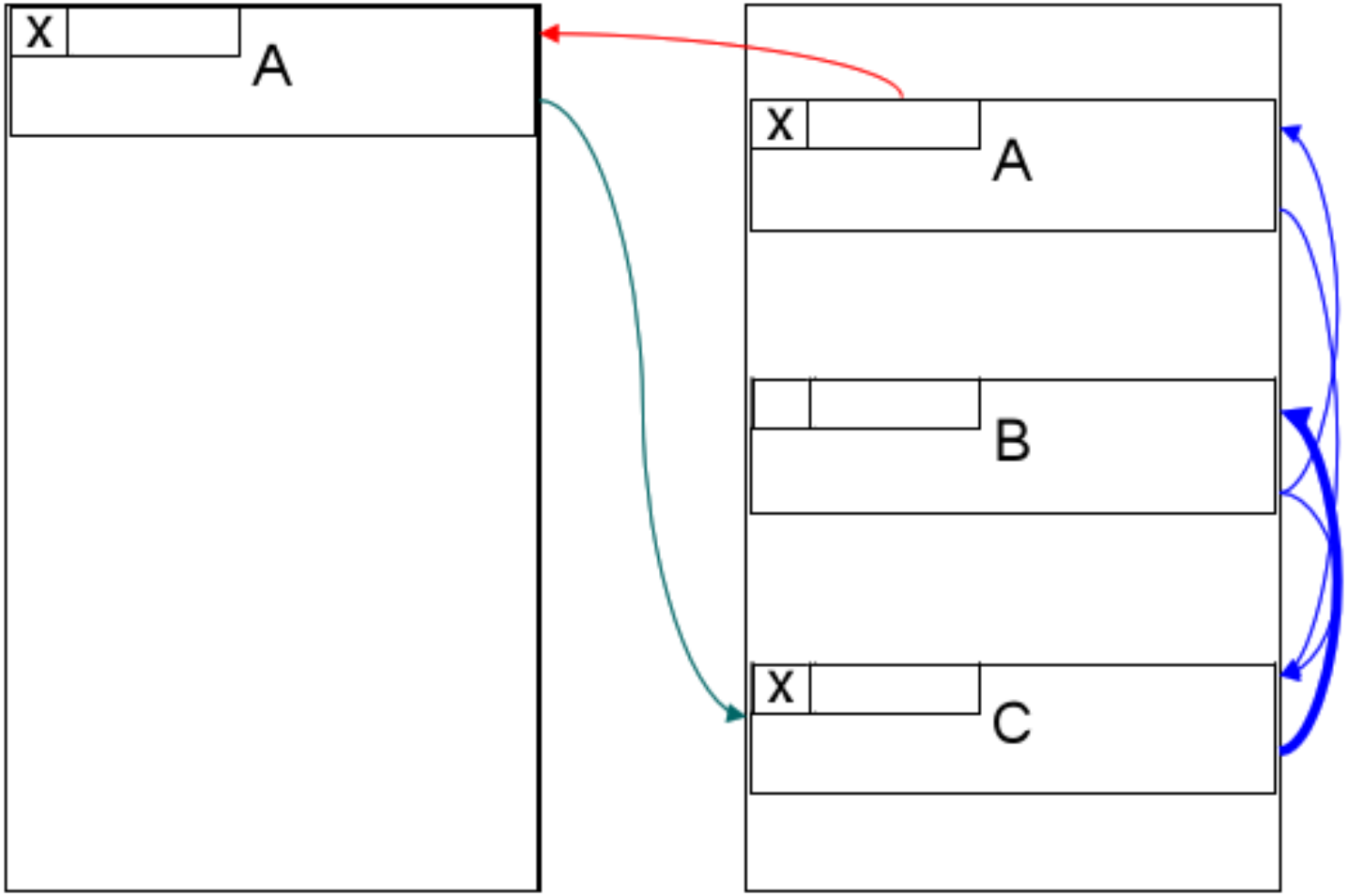


Step by step

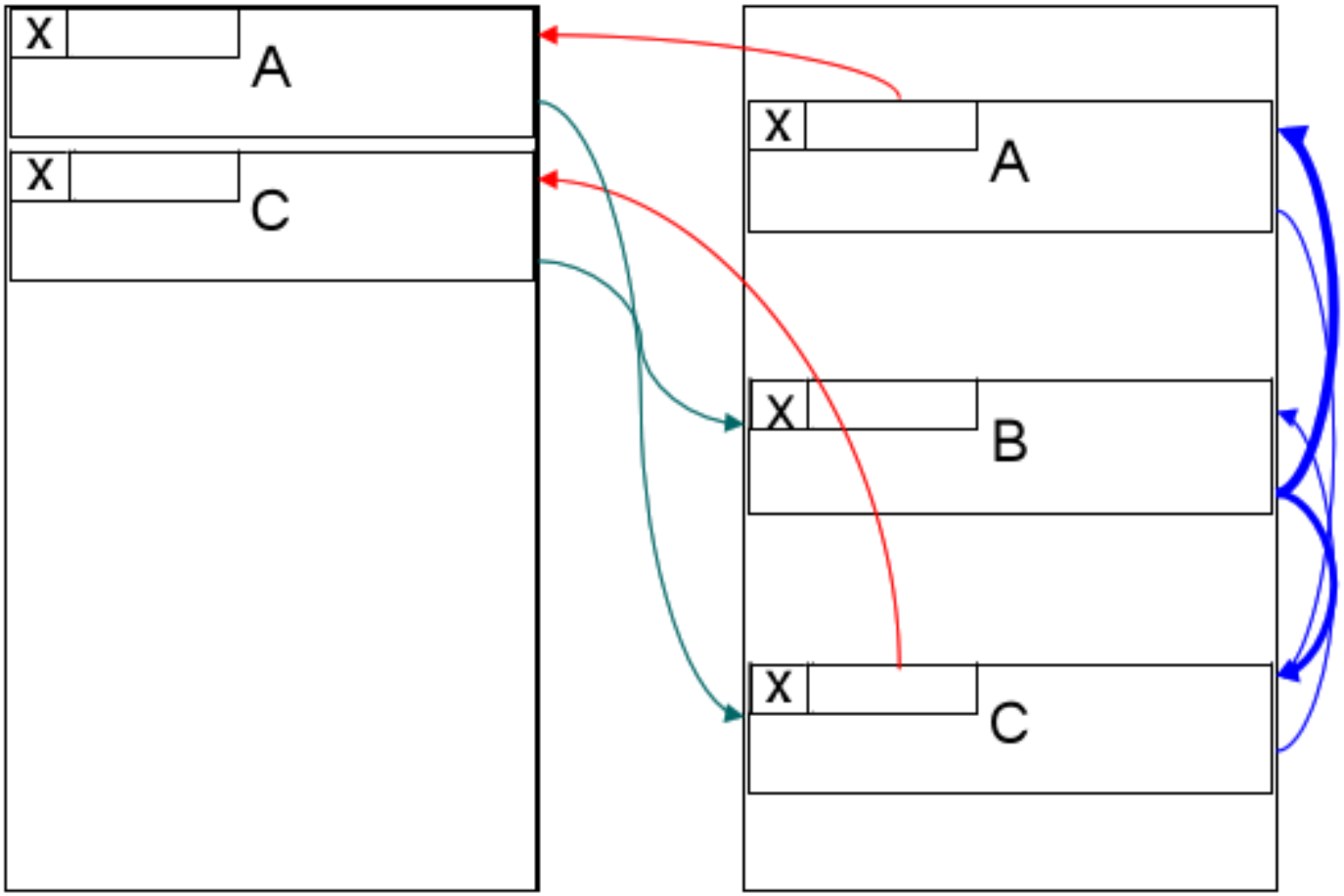




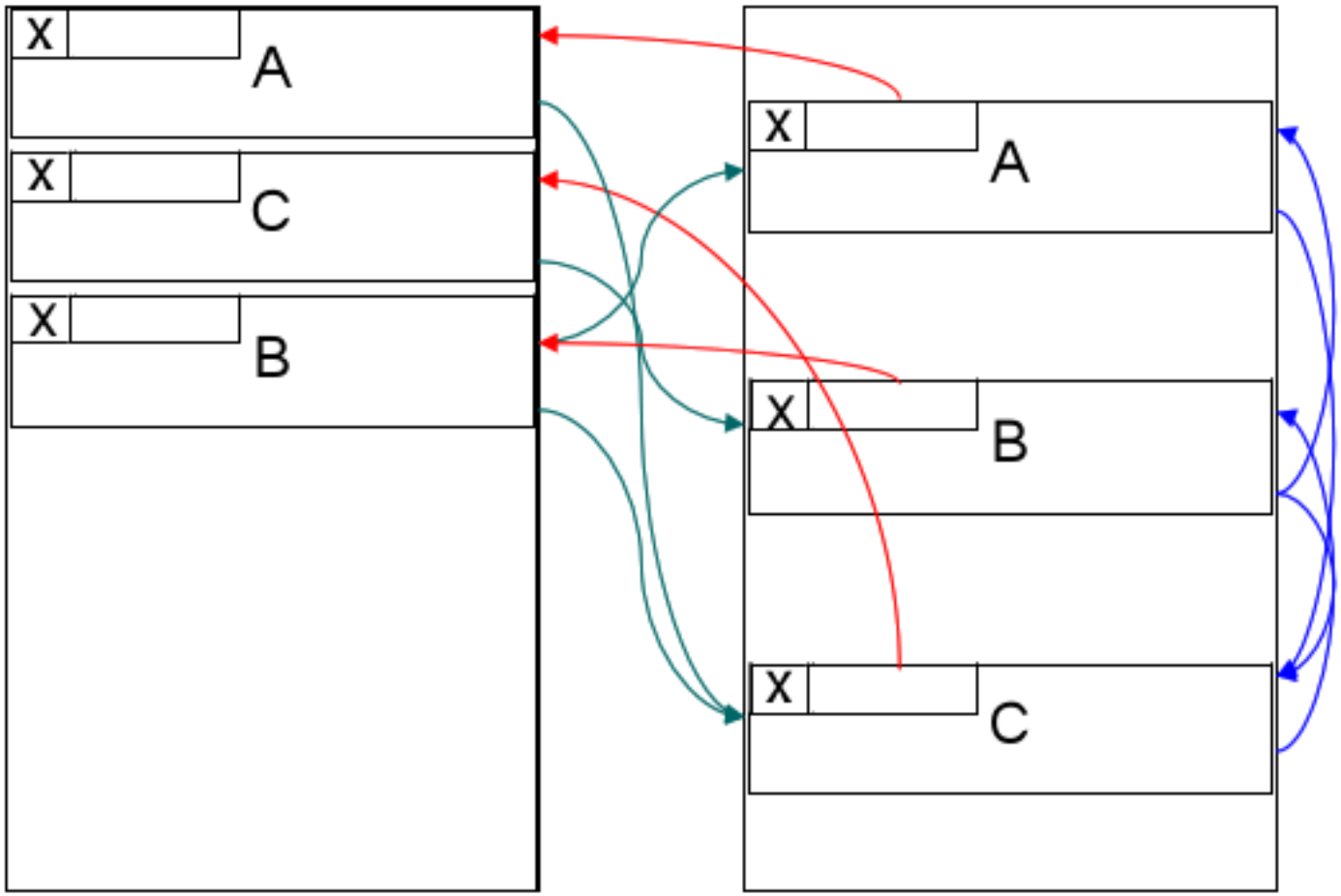
Step by step



Step by step



Step by step



References I



Bibliography

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