Podpora počas behu

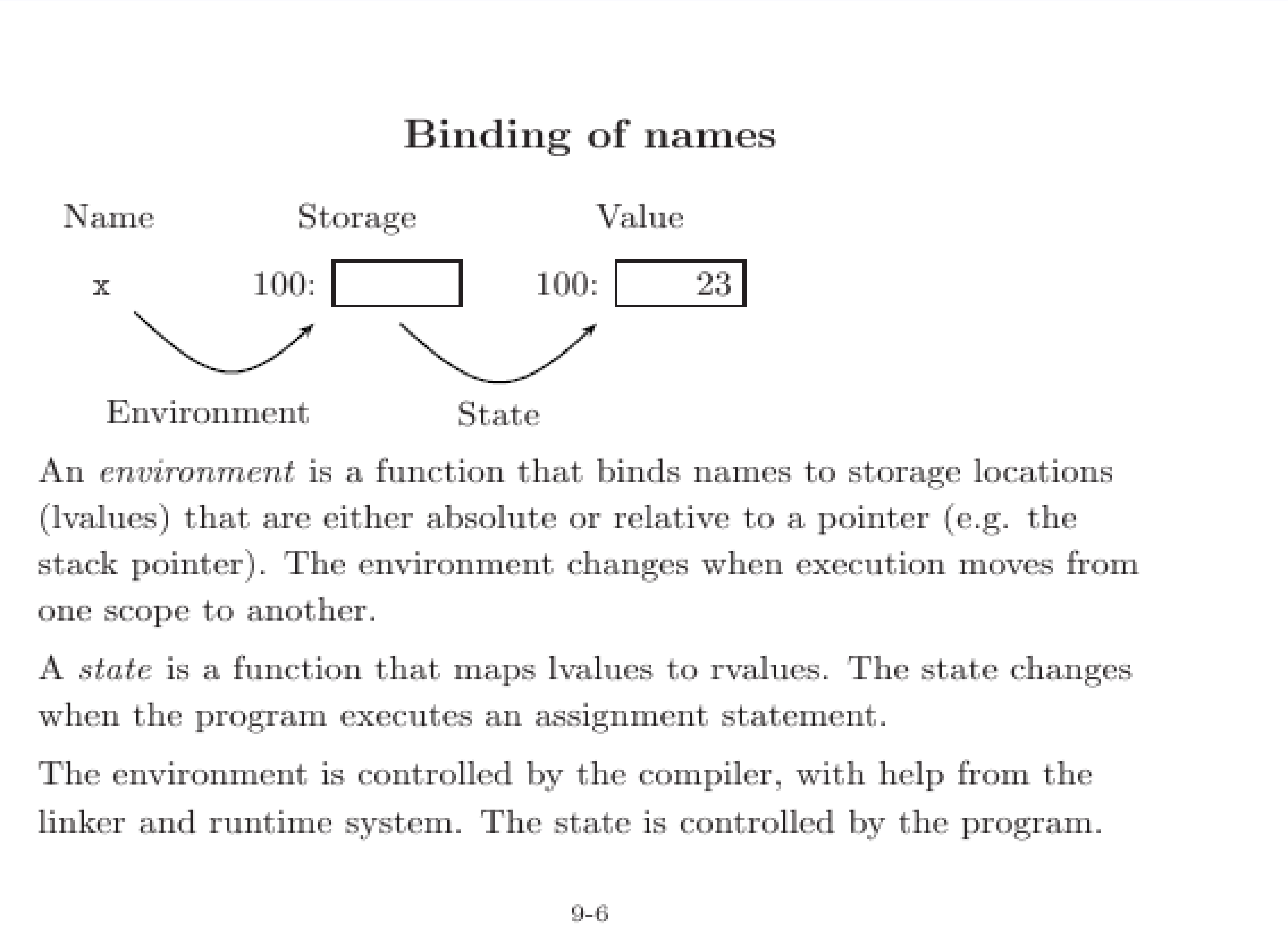
Runtime Environments

Ján Šturc

Zima 2010

Zahrnuje:

- Správu pamäti
- Volanie funkcií a procedúr
- Adresáciu dátových štruktúr
- Štandardné (zabudované funkcie)
- Podporu pre tabuľku symbolov



Binding of names

Name
Storage
Value
x
100:
100:
23
Environment
State

An *environment* is a function that binds names to storage locations (lvalues) that are either absolute or relative to a pointer (e.g. the stack pointer). The environment changes when execution moves from one scope to another.

A *state* is a function that maps lvalues to rvalues. The state changes when the program executes an assignment statement.

The environment is controlled by the compiler, with help from the linker and runtime system. The state is controlled by the program.

Run-Time Environments

Static vs. Runtime

Mapping a high level language to low-level machine environment implies generating code for allocating, maintaining and deallocating data objects to support this activity.

Abstract machine.

Procedures/Functions

- Control Abstraction
  - call/return semantics, parameters, recursion
- Controlled Namespace
  - Scope (local/non-local), binding, addressing
- External Interface
  - separate compilation, libraries (not dealing with here)

Example

```
var a: array [0 .. 10] of integer;
procedure readarray
var i: integer
begin … a[i] … end
function partition(y,z: integer): integer
 var i,j,x,v: integer
 begin … end
 procedure quicksort(m,n: integer)
 var i: integer
 begin i := partition(m,n); quicksort(m,i-1); quicksort(i+1,n) end
procedure main
begin readarray(); quicksort(1,9); end
```

Procedure Call Issues

- Control flow (call and return)
- Data: (formal/actual) parameters and return values
- Scope information
- Responsibilities of caller vs. callee
- Recursion
- Variable addressing

Call Graphs

A **call graph** is a directed multi-graph where:

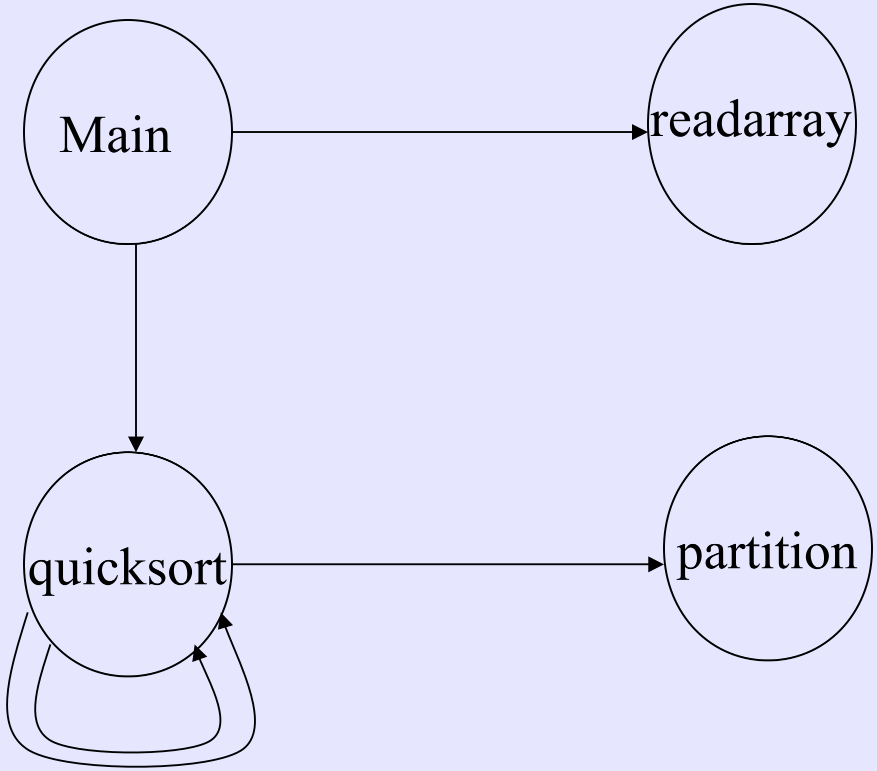
- the nodes are the procedures of the program and
- the edges represent calls between these procedures.

Used in optimization phase.

Acyclic → no recursion in the program

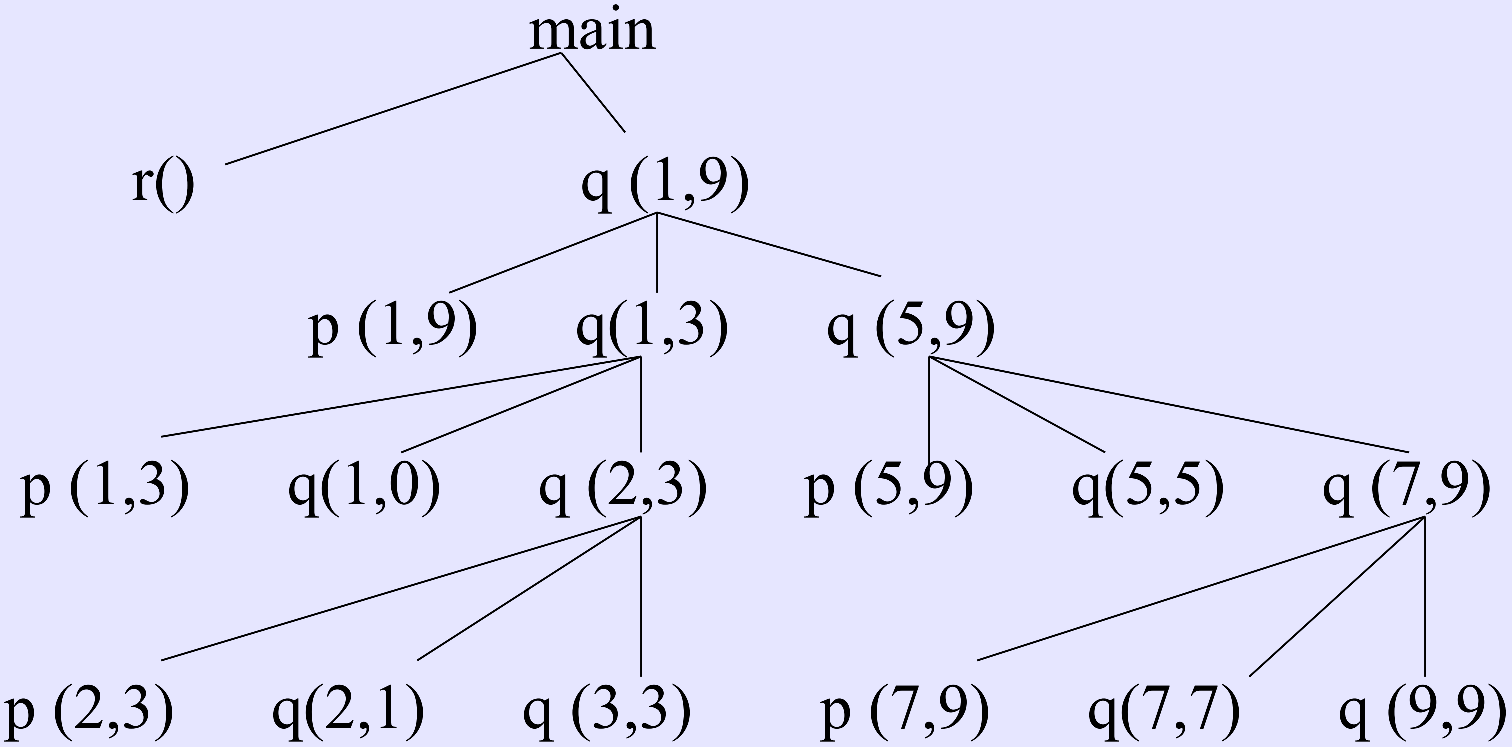
Can be computed **statically**.

Call Graph for Example



Run-time Control Flow

Call Tree - cannot be computed statically



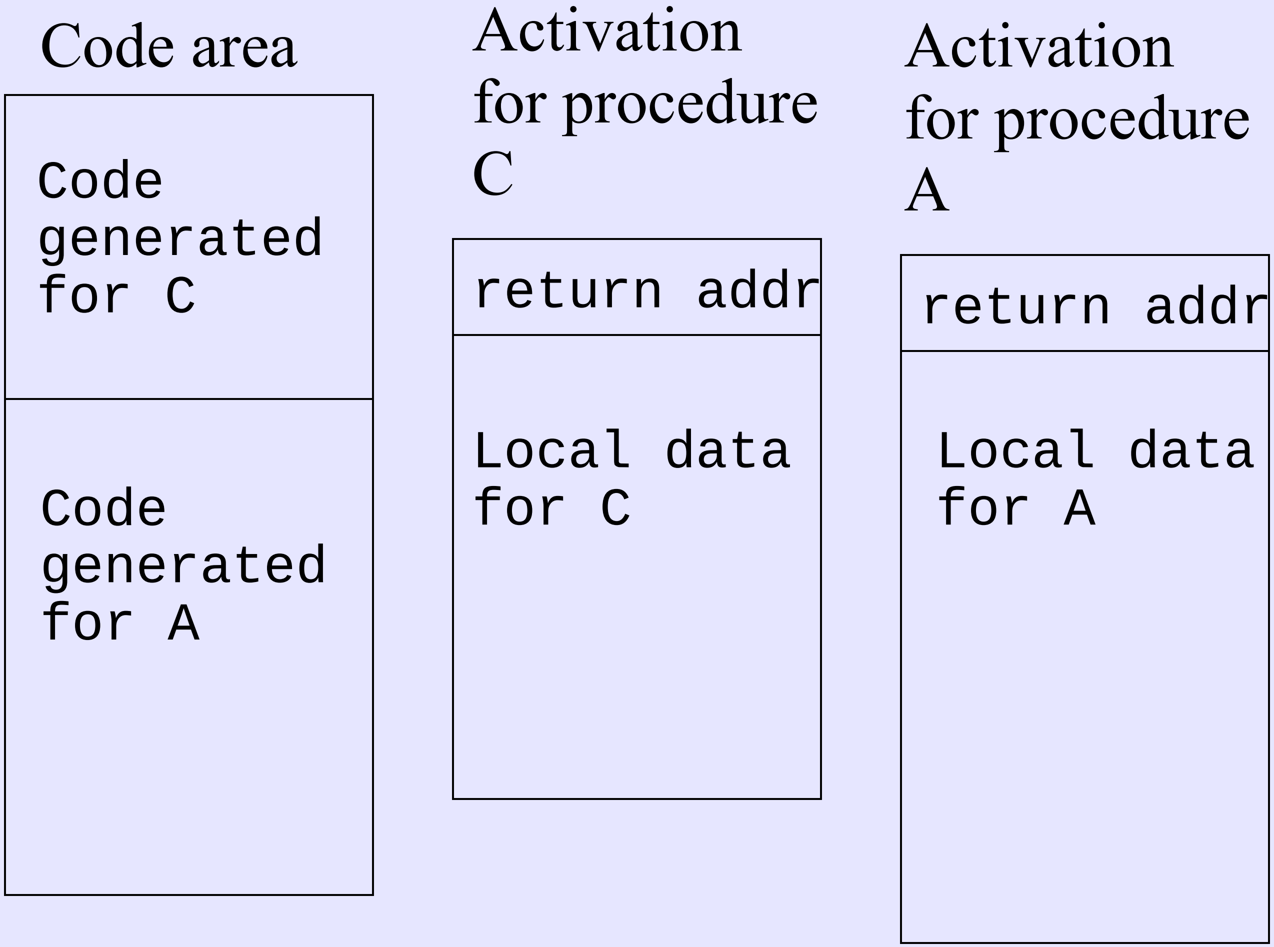
Static Allocation

All space allocated at compile time.

- Code area – machine instructions for each procedure
- Static area –
  - single data area allocated for each procedure.
    - local vars, parameters, return value
  - return address for each procedure.

No recursion

Static Allocation



Code area

Code generated for C

Code generated for A

Activation for procedure C

return addr

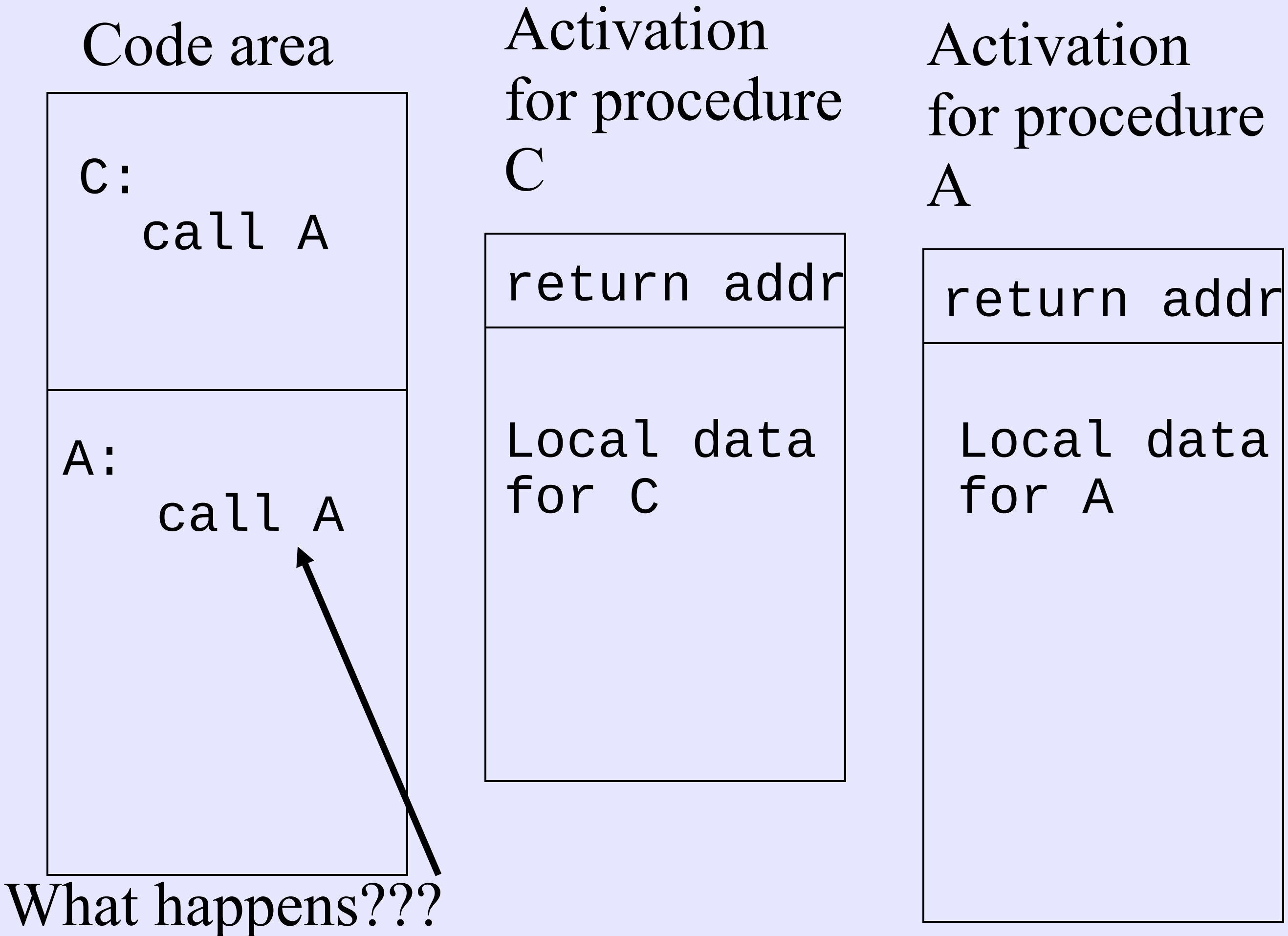
Local data for C

Activation for procedure A

return addr

Local data for A

Static Allocation



Code area

```
C:
  call A
```

```
A:
   call A
```

Activation for procedure C

return addr

Local data for C

Activation for procedure A

return addr

Local data for A

What happens???

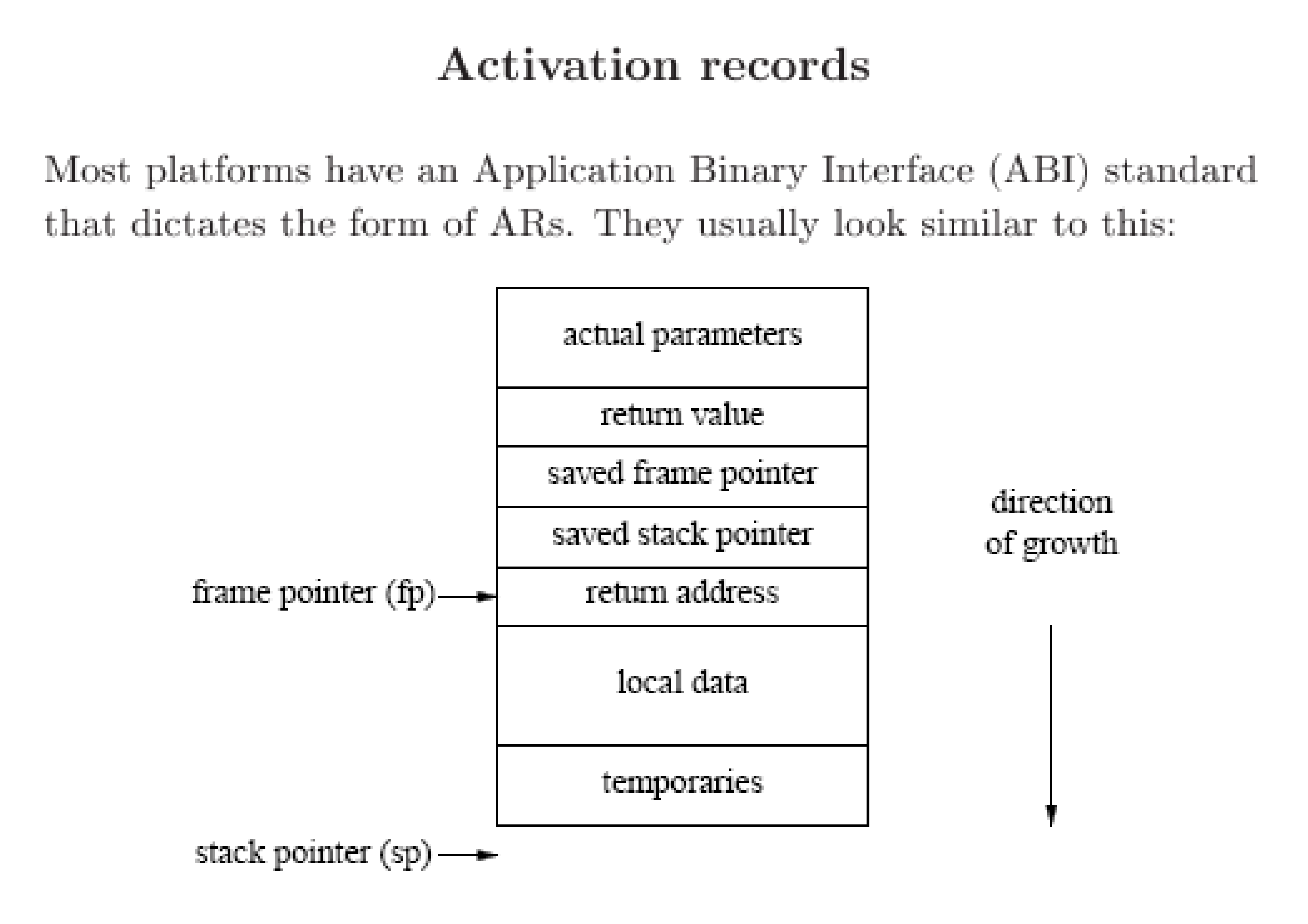
Stack Allocation

- Code area – machine code for procedures
- Static data – not associated with procedures
- Stack – runtime information
  - Return addresses, scope information
- Activation records – allocated at call time onto a runtime stack.

Activation Records

Information needed by a single instance of a procedure.

- Local data
- Parameter storage
- Return value storage
- Control links for stack
- Return address



Activation records

Most platforms have an Application Binary Interface (ABI) standard that dictates the form of ARs. They usually look similar to this:

actual parameters

return value

saved frame pointer

saved stack pointer

frame pointer (fp) → return address

local data

temporaries

stack pointer (sp) →

direction of growth

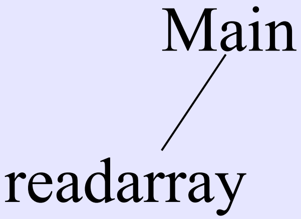
Activation Records

Different procedures/functions will have different size activation records.

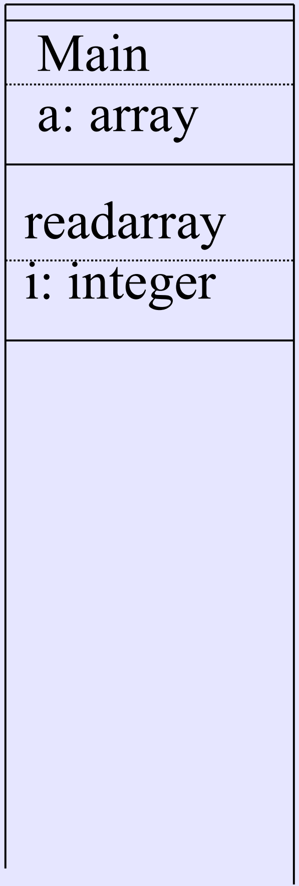
Activation record size can be determined at compile time.

Stack Allocation - 1

Call Tree

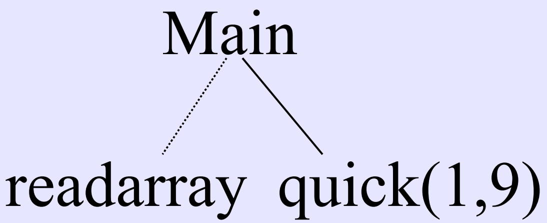


Stack (growing downward)

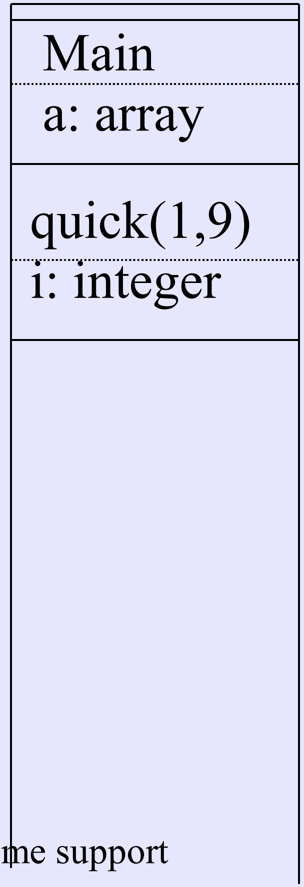


Stack Allocation - 2

Call Tree

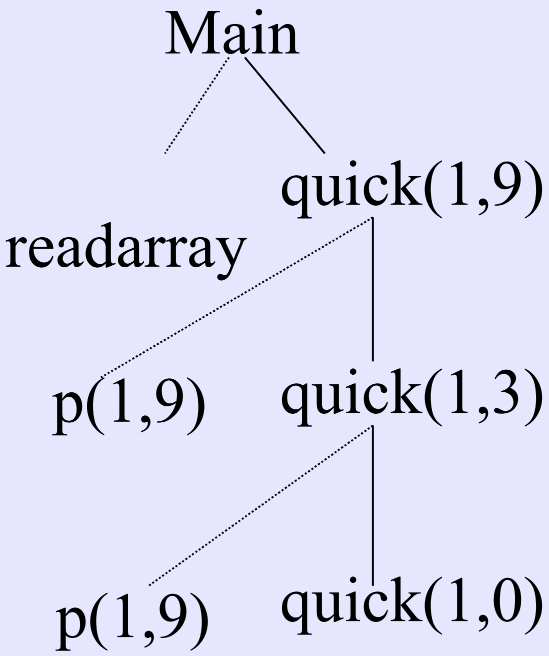


Stack (growing downward)



Stack Allocation - 3

Call Tree



Main

readarray

quick(1,9)

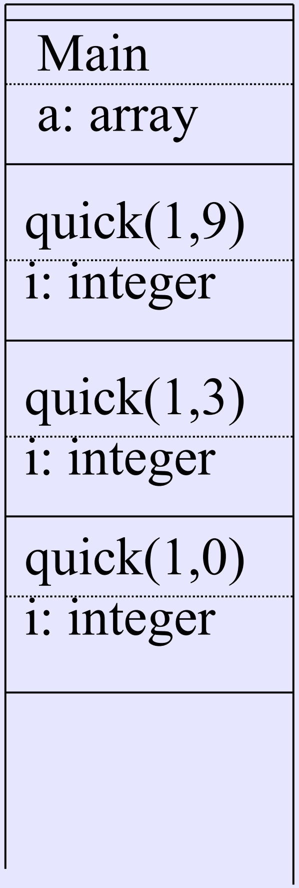
p(1,9)

quick(1,3)

p(1,9)

quick(1,0)

Stack (growing downward)



Main
a: array

quick(1,9)
i: integer

quick(1,3)
i: integer

quick(1,0)
i: integer

Call Processing: Caller

- Create new (callee) activation record on stack.
- Evaluates actual parameters and places them in activation.
- Save registers (→callee) and other status data
- Stores a return address, dynamic link (= current stack pointer), and static link information into callee activation then
- Make stack pointer (SP) to point at new activation.
- Updates the program counter (PC) to the code area for the called procedure.

Added at the point of the call

Call Processing: Callee

- Initializes local data, including moving parameters
- Begins local execution

Added at the start of the function

Return Processing: Callee

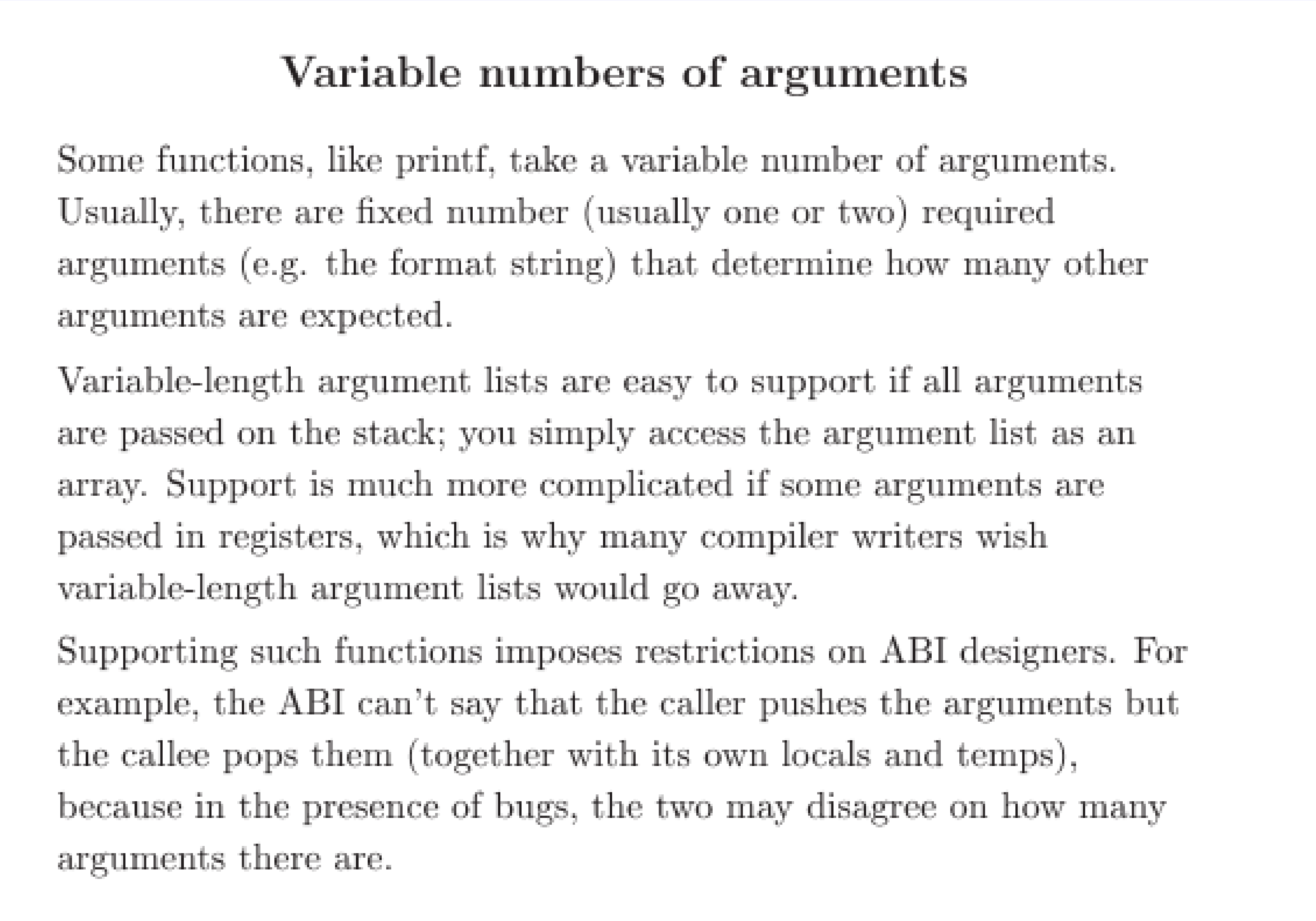
- Place return value (if any) in activation.
  - May be in to accumulator
- Restore SP and PC.

Added at the ‘return’ point(s) of the function

Return Processing: Caller

- Restore registers and status
- Copy the return value (if any) from activation
- Continue local execution

Added after the point of the call



Variable numbers of arguments

Some functions, like printf, take a variable number of arguments. Usually, there are fixed number (usually one or two) required arguments (e.g. the format string) that determine how many other arguments are expected.

Variable-length argument lists are easy to support if all arguments are passed on the stack; you simply access the argument list as an array. Support is much more complicated if some arguments are passed in registers, which is why many compiler writers wish variable-length argument lists would go away.

Supporting such functions imposes restrictions on ABI designers. For example, the ABI can't say that the caller pushes the arguments but the callee pops them (together with its own locals and temps), because in the presence of bugs, the two may disagree on how many arguments there are.

Runtime Addressing

- Given a variable reference in the code, how can we find the correct instance of that variable?
- Tied to issues of scope

Scope

The **scope** of a variable is that portion of the programs to which the variable applies.

- A variable is **local** to a procedure if the declaration occurs in that procedure.
- A variable is **non-local** to a procedure if it is not local to that procedure but the declaration occurs occurs in an enclosing scope of that procedure.
- A variable is **global** if it occurs in the outermost scope.

Types of Scoping

- Static – scope of a variable determined from the source code. Scope A is enclosed in scope B if A's source code is nested inside B's source code.
- Dynamic – current call tree determines the relevant declaration of a variable use.

Most Closely Nested Rule

The scope of a particular declaration is given by the most closely nested rule

- The scope of a variable declared in block B, includes B.
- If x is not declared in block B, then an occurrence of x in B is in the scope of a declaration of x in some enclosing block A, such that A has a declaration of x and A is more closely nested around B than any other block with a declaration of x.

Runtime Addressing in Static Allocation

- Variable addresses hard-coded, usually as offset from data area where variable is declared.
  - addr(x) = start of x's local scope + x's offset

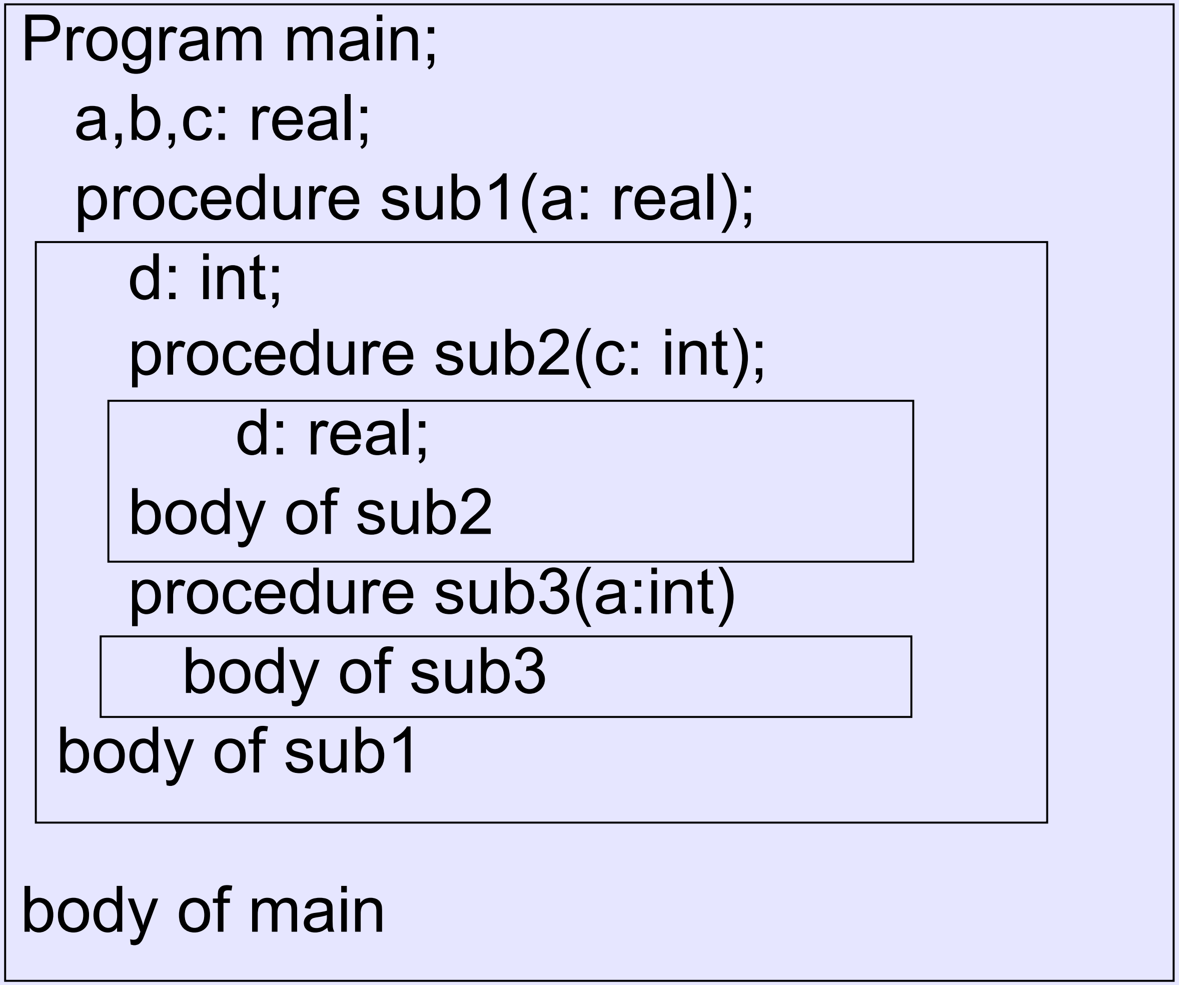
Control Links in Stack Allocation

- Dynamic – points to caller’s activation (old stack pointer)
- Static (access) link – points to enclosing scope
  - If callee is directly enclosed by caller, static link = caller’s activation
  - If callee has same enclosing scope as caller, static link = static link in caller’s activation
  - If callee is in a scope (k levels up) that encloses the caller, need to traverse k static links from the caller and use that static link.

Runtime Addressing in Stack Allocation

- At runtime, we can’t know where the relevant activation record holding the variable exists on the stack
- Use static (access) links to enable quick location
  - addr(x) = # static links + x's offset
  - Local: (0,offset)
  - Immediately enclosing scope: (1,offset)

Example Program



```
Program main;
   a,b,c: real;
   procedure sub1(a: real);
      d: int;
      procedure sub2(c: int);
            d: real;
      body of sub2
      procedure sub3(a:int)
         body of sub3
  body of sub1

body of main
```

Variables from Example

Procedure	**Enclosing**	**Local:addr = offset**	**Non-local: addr =(scope,offset)**
main	-	a:0,b:1,c:2	-
sub1	main	a:0,d:1	b:(main,1),c:(main,2)

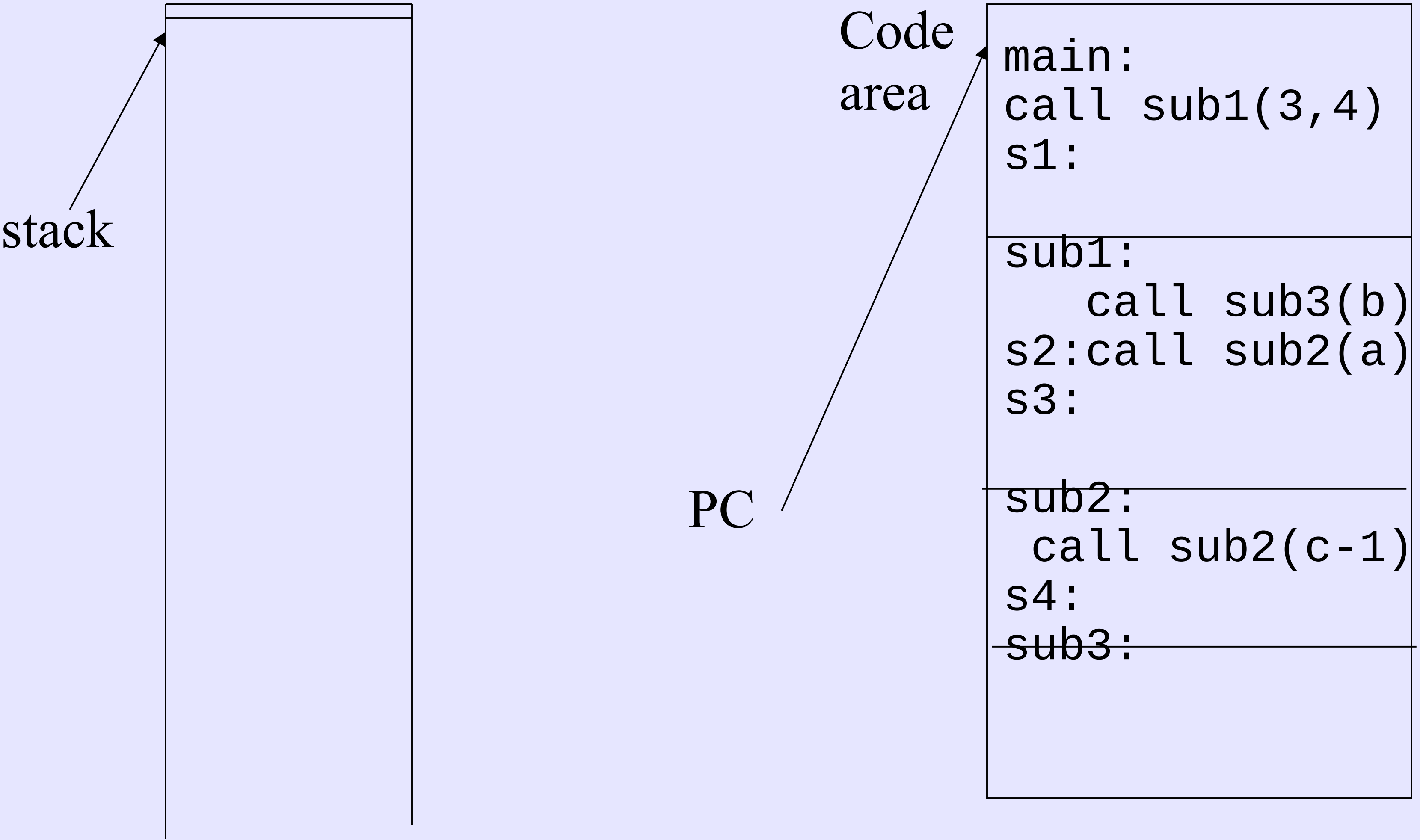
sub2	sub1	c:0,d:1	b:(main,1)
			a:(sub1,0)

sub3	sub2	a:0	b:(main,1),c:(main,2)
			d:(sub1,1)

Example Program

```
Program main;
    procedure sub1(a: int,b:int);
      procedure sub2(c: int);
        if c > 0 call sub2(c-1)
      procedure sub3()
        body of sub3
  call sub3(b);  call sub2(a);
call sub1(3,4);
```

Example Program at runtime 1

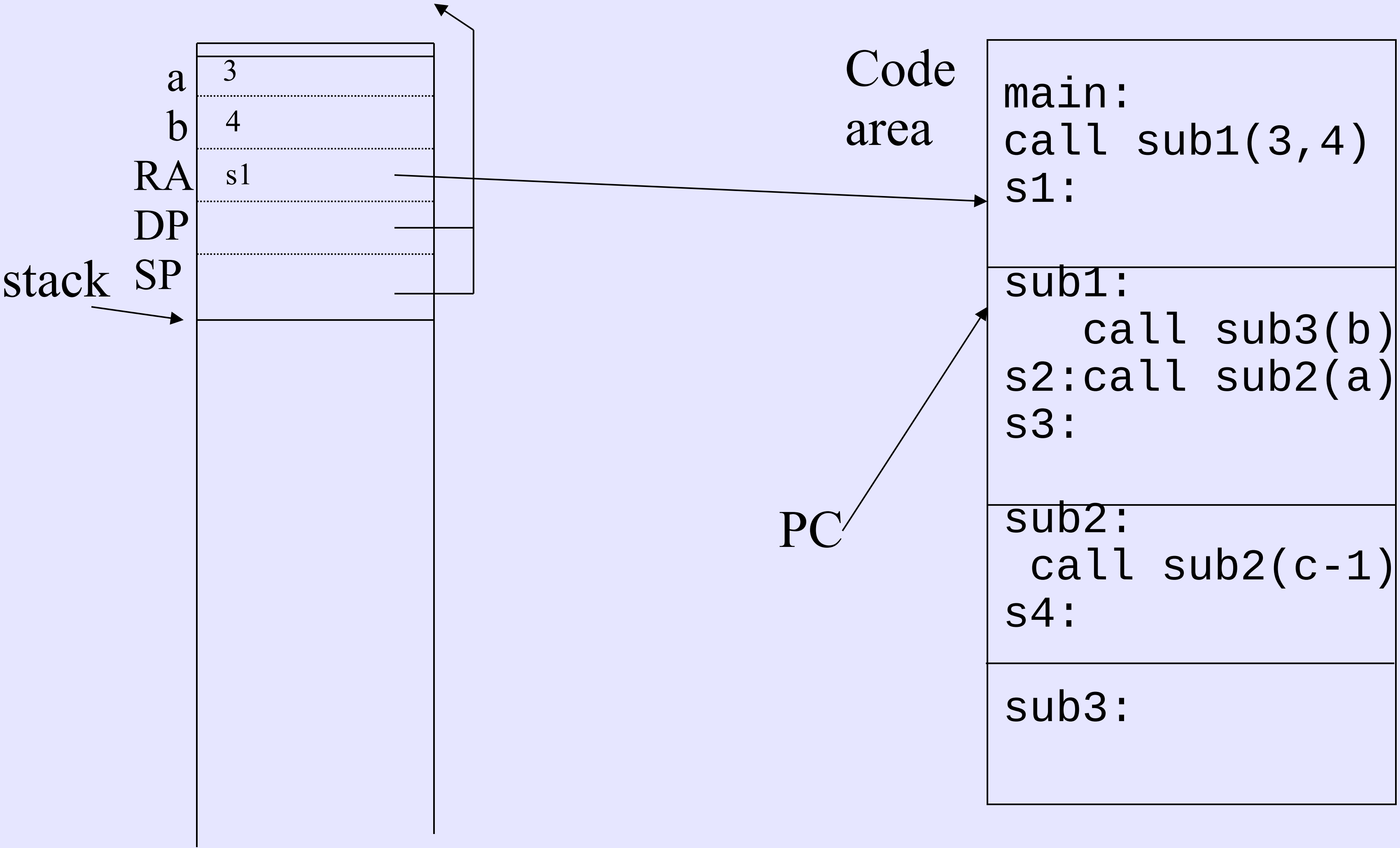


```
main:
call sub1(3,4)
s1:

sub1:
   call sub3(b)
s2:call sub2(a)
s3:

sub2:
 call sub2(c-1)
s4:
sub3:
```

Example Program at runtime 2



a 3

b 4

RA s1

DP

SP

stack

Code area

PC

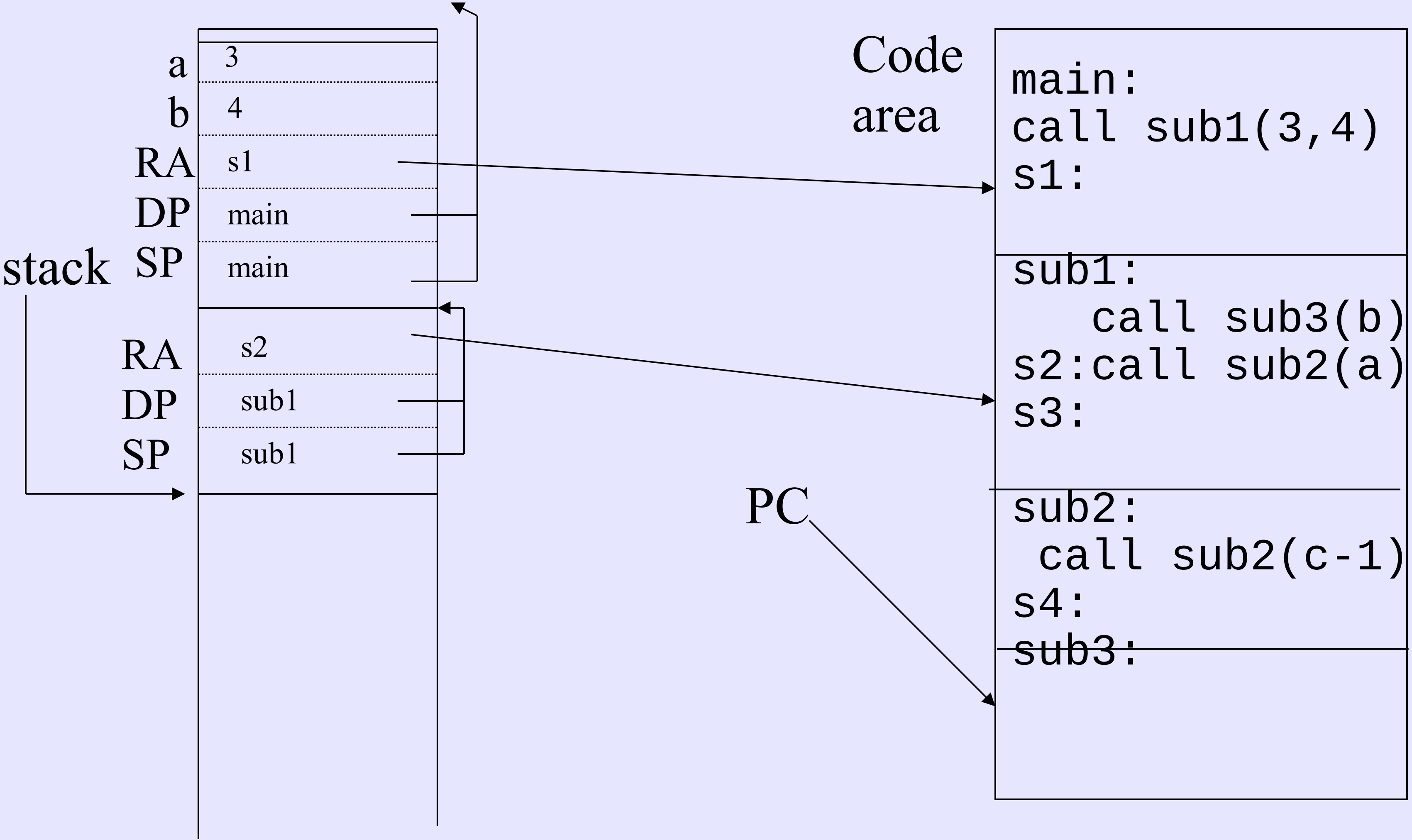
```
main:
call sub1(3,4)
s1:

sub1:
   call sub3(b)
s2:call sub2(a)
s3:

sub2:
 call sub2(c-1)
s4:

sub3:
```

Example Program at runtime 3



stack

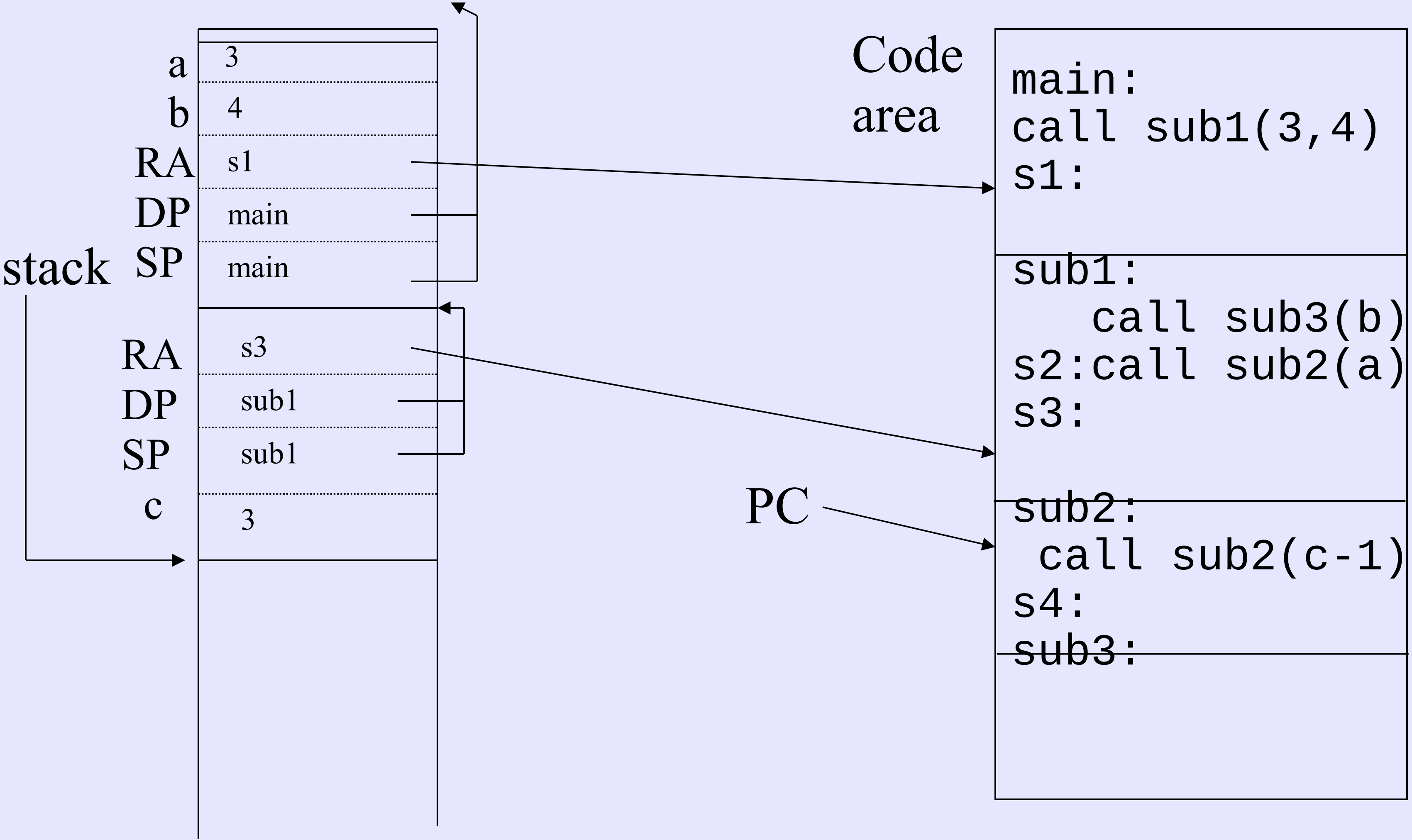
a	3
b	4
RA	s1
DP	main
SP	main
RA	s2
DP	sub1
SP	sub1

Code area

```
main:
call sub1(3,4)
s1:
sub1:
   call sub3(b)
s2:call sub2(a)
s3:
sub2:
 call sub2(c-1)
s4:
sub3:
```

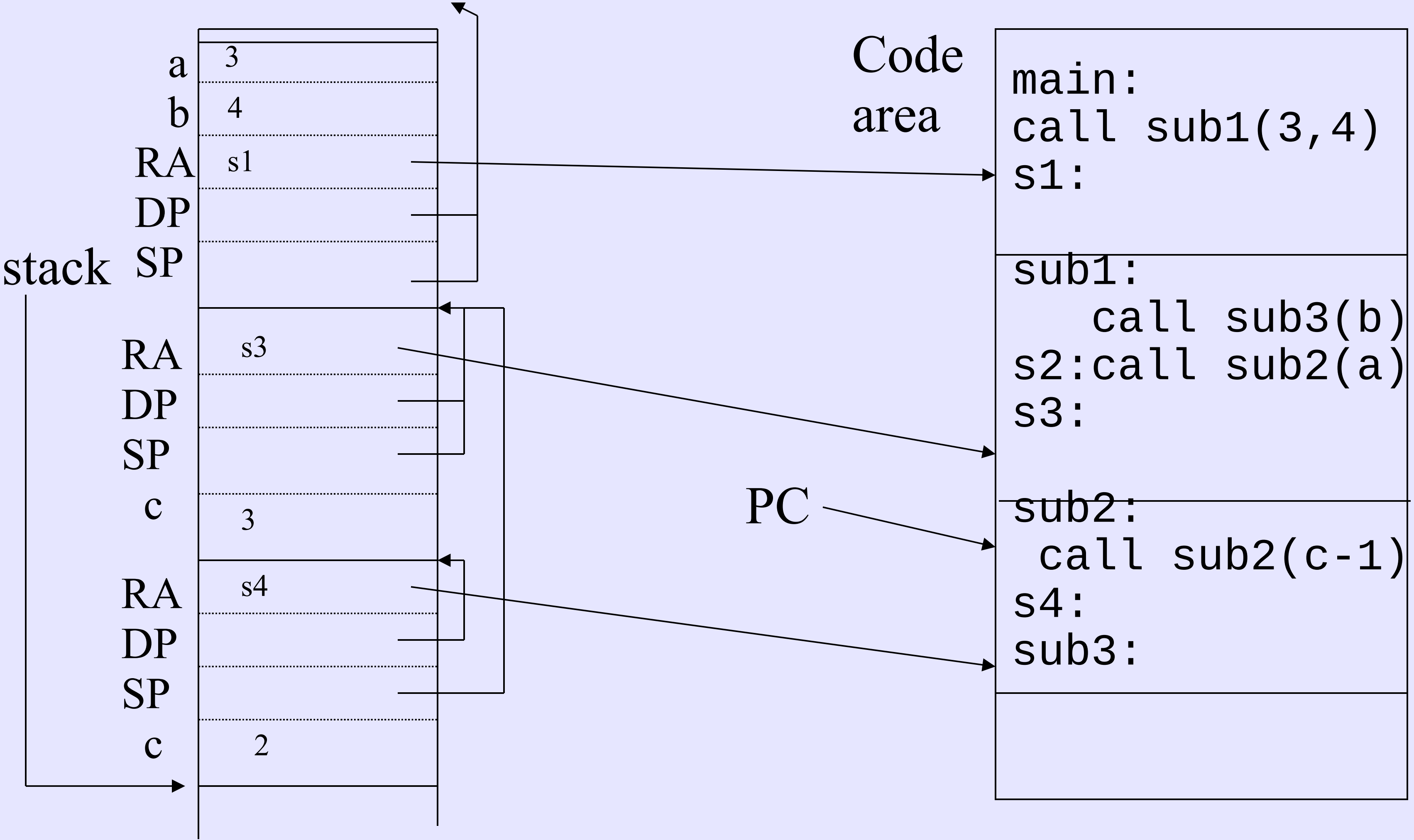
PC

Example Program at runtime 4



```
main:
call sub1(3,4)
s1:
sub1:
   call sub3(b)
s2:call sub2(a)
s3:
sub2:
 call sub2(c-1)
s4:
sub3:
```

Example Program at runtime 5



```
main:
call sub1(3,4)
s1:

sub1:
   call sub3(b)
s2:call sub2(a)
s3:

sub2:
 call sub2(c-1)
s4:
sub3:
```

Display

Alternate representation for access information.

- The current static chain kept in a single location
- Advantages: faster addressing
- Disadvantages: additional data structure to store and maintain.

Parameter Passing

- Call-by-value – data is copied at the callee and any item changes do not affect values in the caller.
- Call-by-reference – pointer to to data is given to the callee and any changes made by the callee are indirect references to the actual value in the caller.
- Call-by-value-result (copy-restore) – hybrid of call-by-value and call-by-reference. Data copied at the callee. During the call, changes do not affect the actual parameter. After the call, the actual value is updated.
- Call-by-name – the actual parameter is in-line substituted into the called procedure. This means it is not evaluated until it is used.

Call-by-value vs. Call-by-reference

```
var a,b : integer
    procedure swap(x,y : integer);
    var t: integer;
    begin  t := x; x := y; y := t;  end;
begin
a := 1;  b := 2;
swap(a,b);
write ('a = ',a);
write ('b = ',b);
end.
```

	value	reference
write(a)	1	2
write(b)	2	1

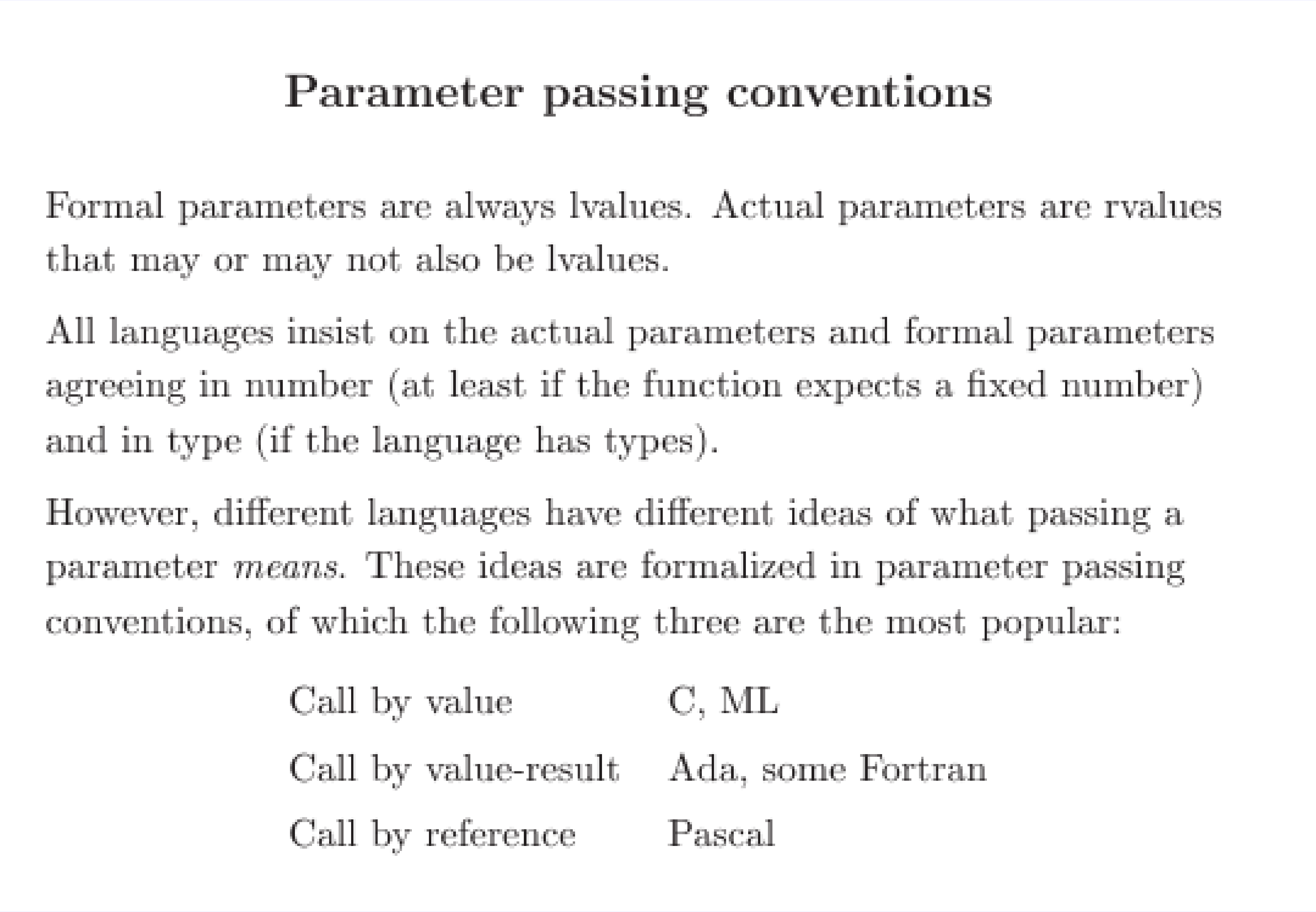
Call-by-value-result vs. Call-by-reference

```
var a: integer
   procedure foo(x: integer);
   begin  a := a + 1;  x := x + 1;  end;
begin
a := 1;
foo(a);
write ('a = ',a);
end.
```

	Value-result	reference
write(a)	2	3

Call-by-name vs Call-by-reference

```
var a : array of integer; i: integer
    procedure swap(x,y : integer);
    var t: integer;
    begin  t := x; x := y; y := t;  end;
begin
i := 1;
swap(i,a[i]);
write (‘i,a[i] = ',i,a[i]);
end.
```



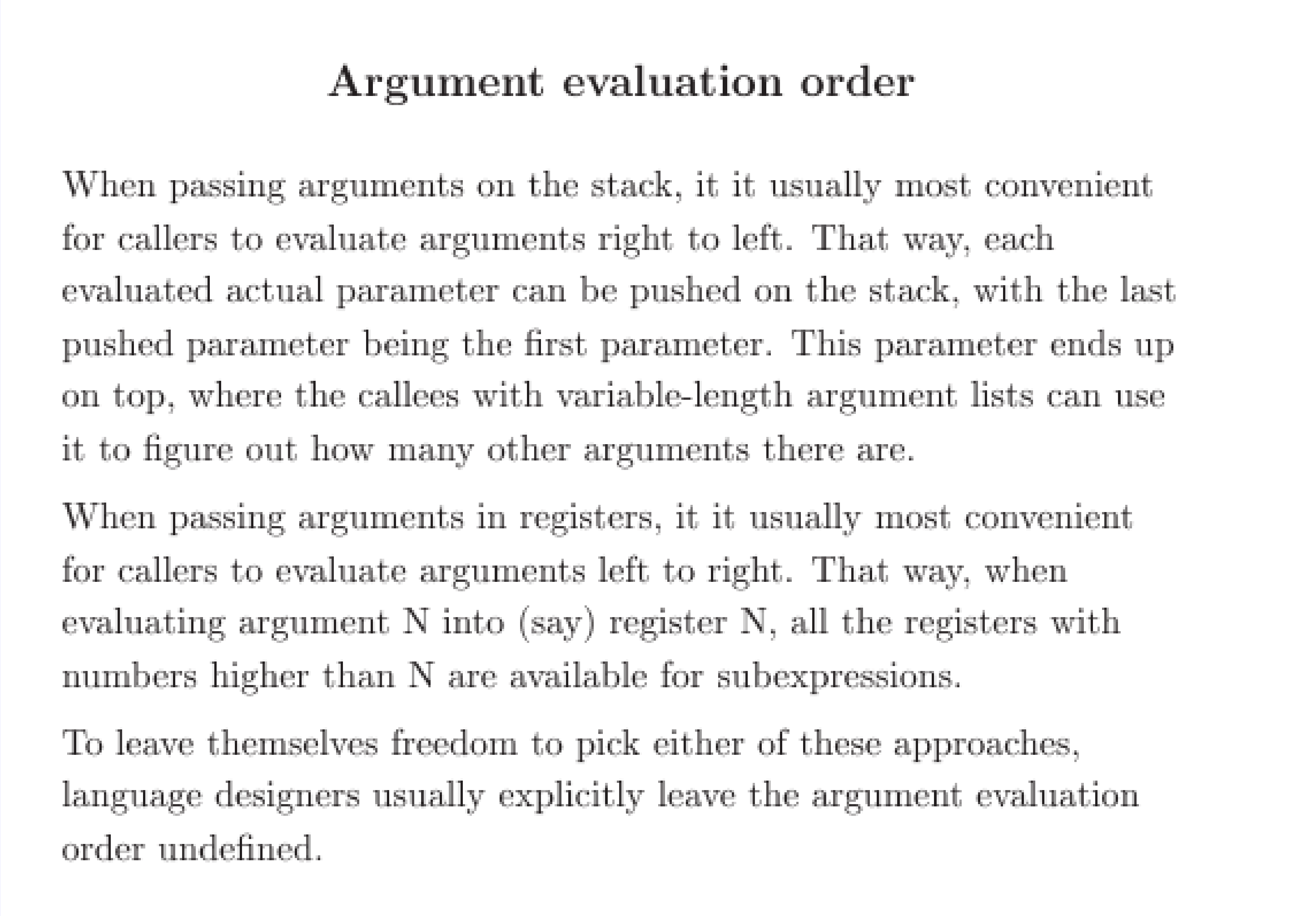
Parameter passing conventions

Formal parameters are always lvalues. Actual parameters are rvalues that may or may not also be lvalues.

All languages insist on the actual parameters and formal parameters agreeing in number (at least if the function expects a fixed number) and in type (if the language has types).

However, different languages have different ideas of what passing a parameter *means*. These ideas are formalized in parameter passing conventions, of which the following three are the most popular:

Call by value	C, ML
Call by value-result	Ada, some Fortran
Call by reference	Pascal



Argument evaluation order

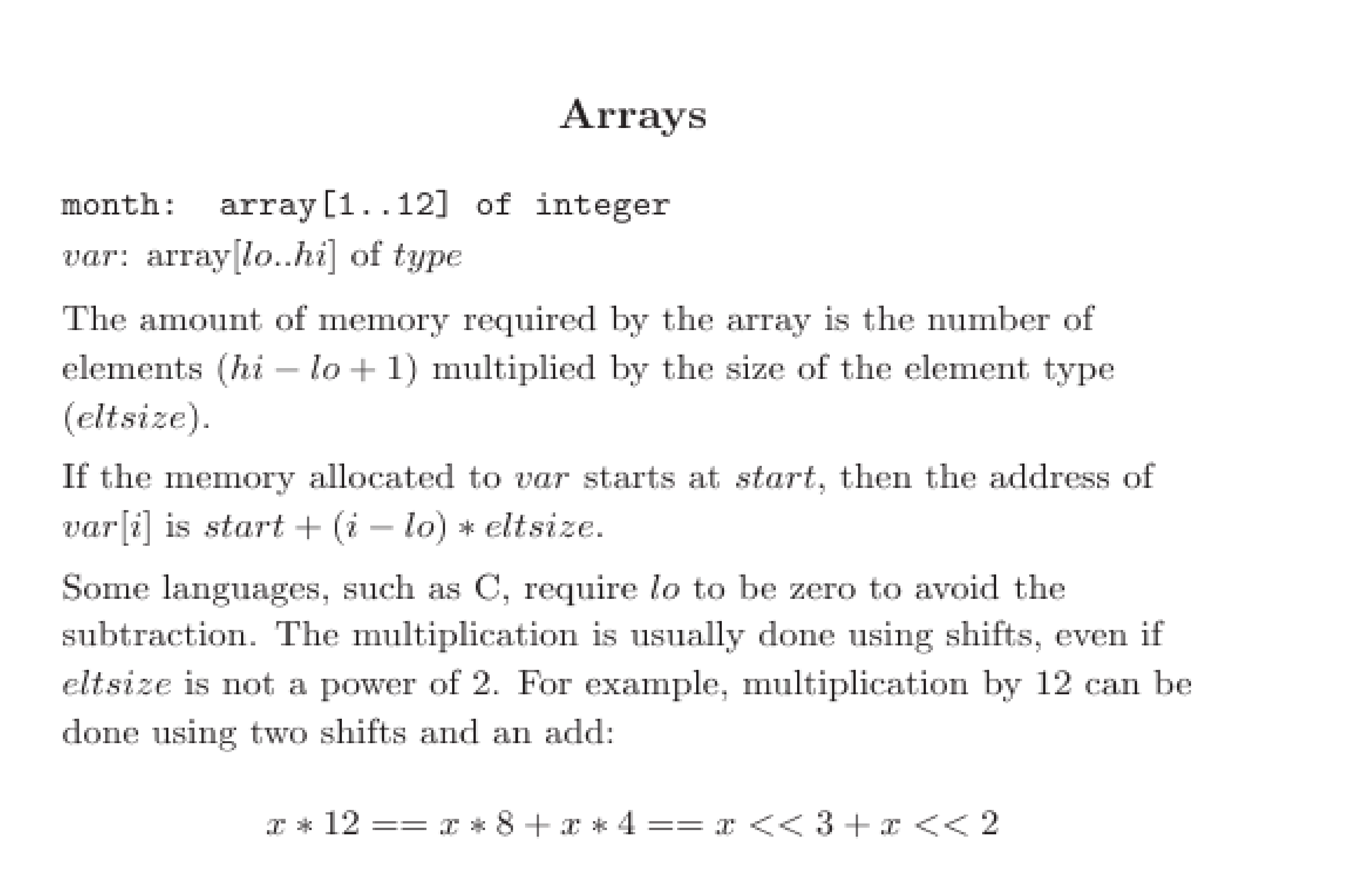
When passing arguments on the stack, it it usually most convenient for callers to evaluate arguments right to left. That way, each evaluated actual parameter can be pushed on the stack, with the last pushed parameter being the first parameter. This parameter ends up on top, where the callees with variable-length argument lists can use it to figure out how many other arguments there are.

When passing arguments in registers, it it usually most convenient for callers to evaluate arguments left to right. That way, when evaluating argument N into (say) register N, all the registers with numbers higher than N are available for subexpressions.

To leave themselves freedom to pick either of these approaches, language designers usually explicitly leave the argument evaluation order undefined.

Adresovanie polí

- Po riadkoch
- Po stĺpcoch
- Od prvého elementu
- Od fiktívnej adresy A [0, …, 0]
- Oba spôsoby sa dajú efektívne optimalizovať



Arrays

```
month:  array[1..12] of integer
```

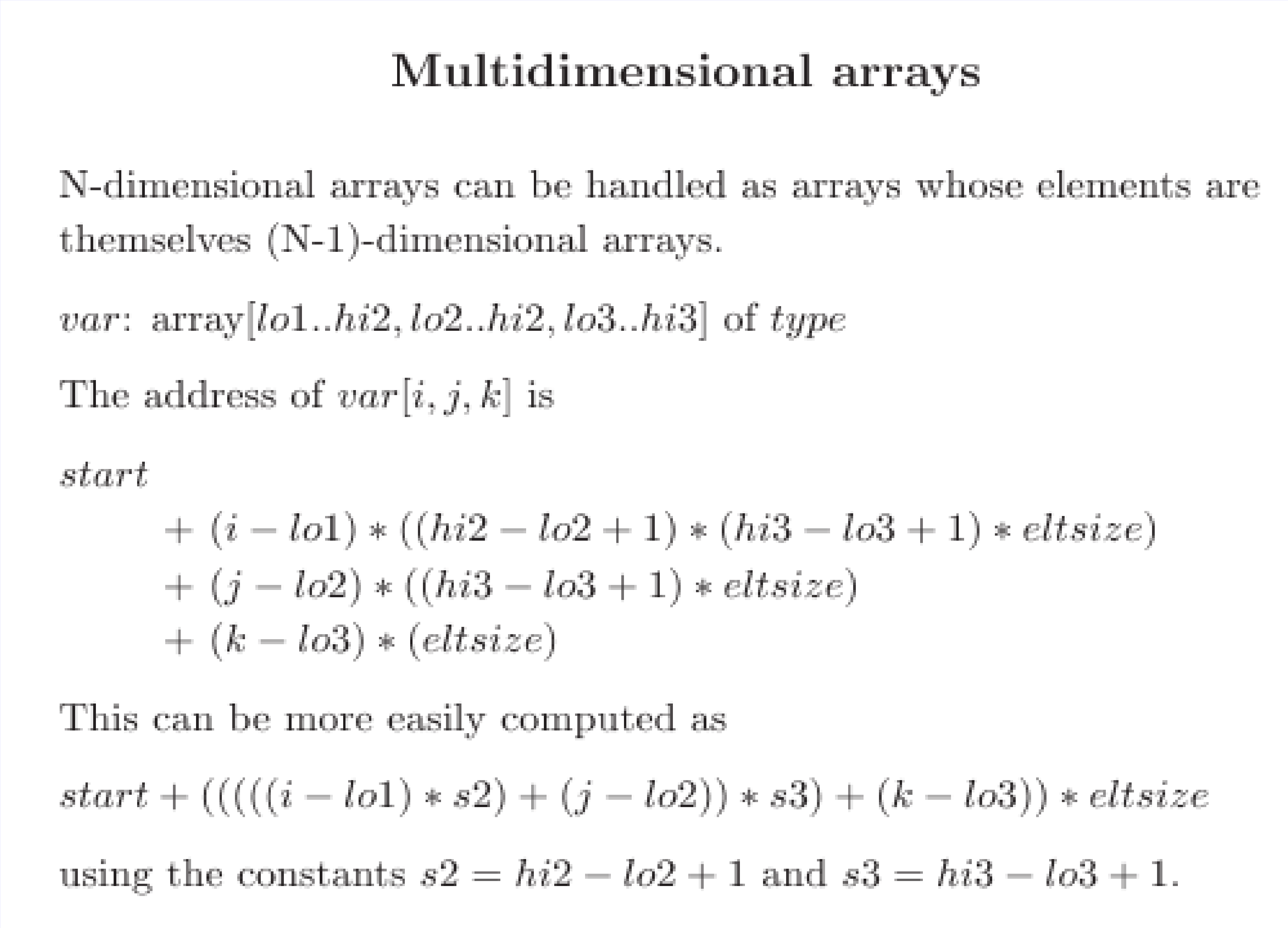
var: array[*lo*..*hi*] of *type*

The amount of memory required by the array is the number of elements $(hi - lo + 1)$ multiplied by the size of the element type $(eltsize)$.

If the memory allocated to *var* starts at *start*, then the address of $var[i]$ is $start + (i - lo) * eltsize$.

Some languages, such as C, require *lo* to be zero to avoid the subtraction. The multiplication is usually done using shifts, even if *eltsize* is not a power of 2. For example, multiplication by 12 can be done using two shifts and an add:

$$x * 12 == x * 8 + x * 4 == x << 3 + x << 2$$



Multidimensional arrays

N-dimensional arrays can be handled as arrays whose elements are themselves (N-1)-dimensional arrays.

var: array$[lo1..hi2, lo2..hi2, lo3..hi3]$ of $type$

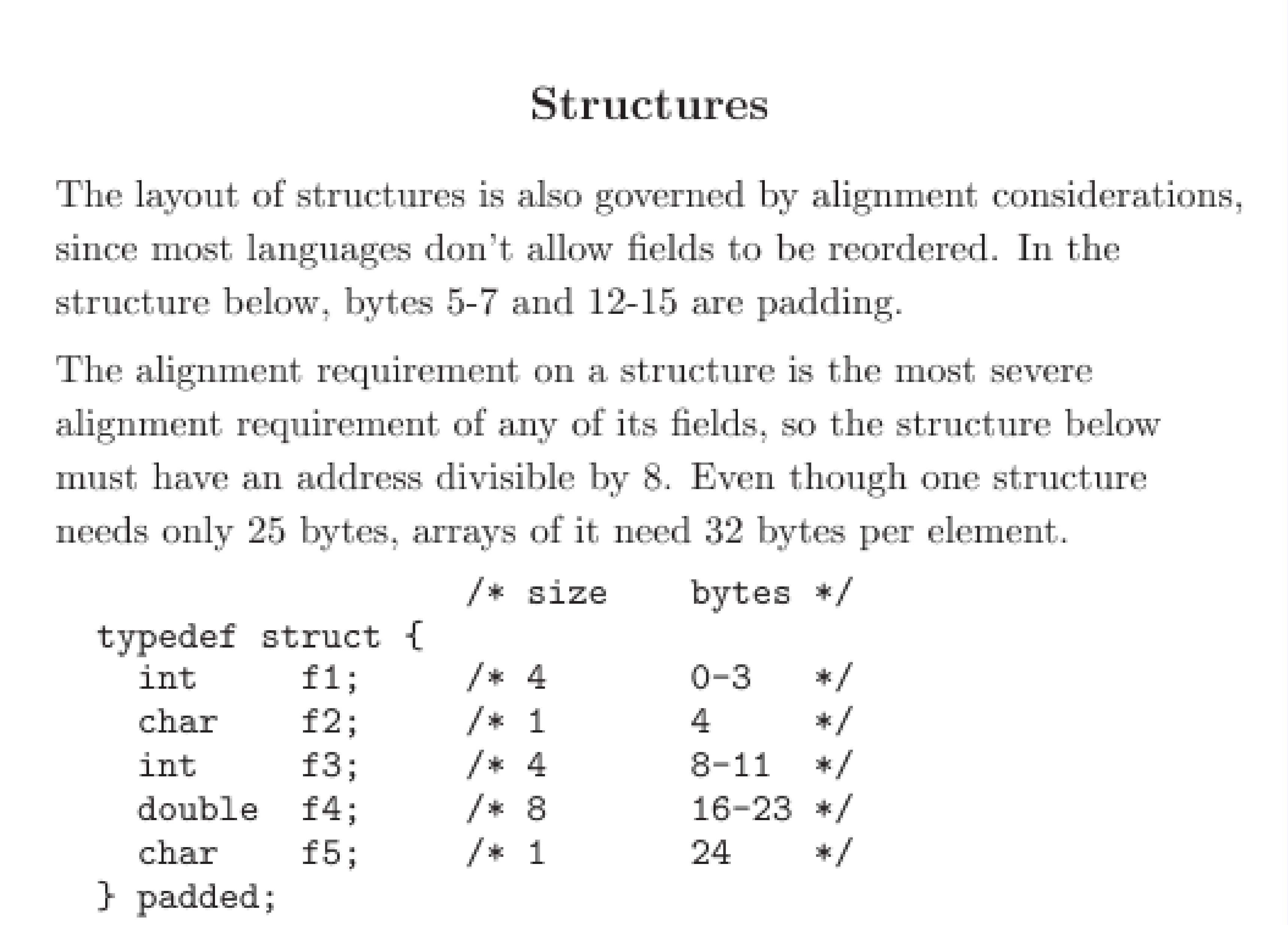
The address of $var[i, j, k]$ is

$$\begin{aligned}
&start\\
&\qquad + (i - lo1) * ((hi2 - lo2 + 1) * (hi3 - lo3 + 1) * eltsize)\\
&\qquad + (j - lo2) * ((hi3 - lo3 + 1) * eltsize)\\
&\qquad + (k - lo3) * (eltsize)
\end{aligned}$$

This can be more easily computed as

$$start + (((((i - lo1) * s2) + (j - lo2)) * s3) + (k - lo3)) * eltsize$$

using the constants $s2 = hi2 - lo2 + 1$ and $s3 = hi3 - lo3 + 1$.



Structures

The layout of structures is also governed by alignment considerations, since most languages don't allow fields to be reordered. In the structure below, bytes 5-7 and 12-15 are padding.

The alignment requirement on a structure is the most severe alignment requirement of any of its fields, so the structure below must have an address divisible by 8. Even though one structure needs only 25 bytes, arrays of it need 32 bytes per element.

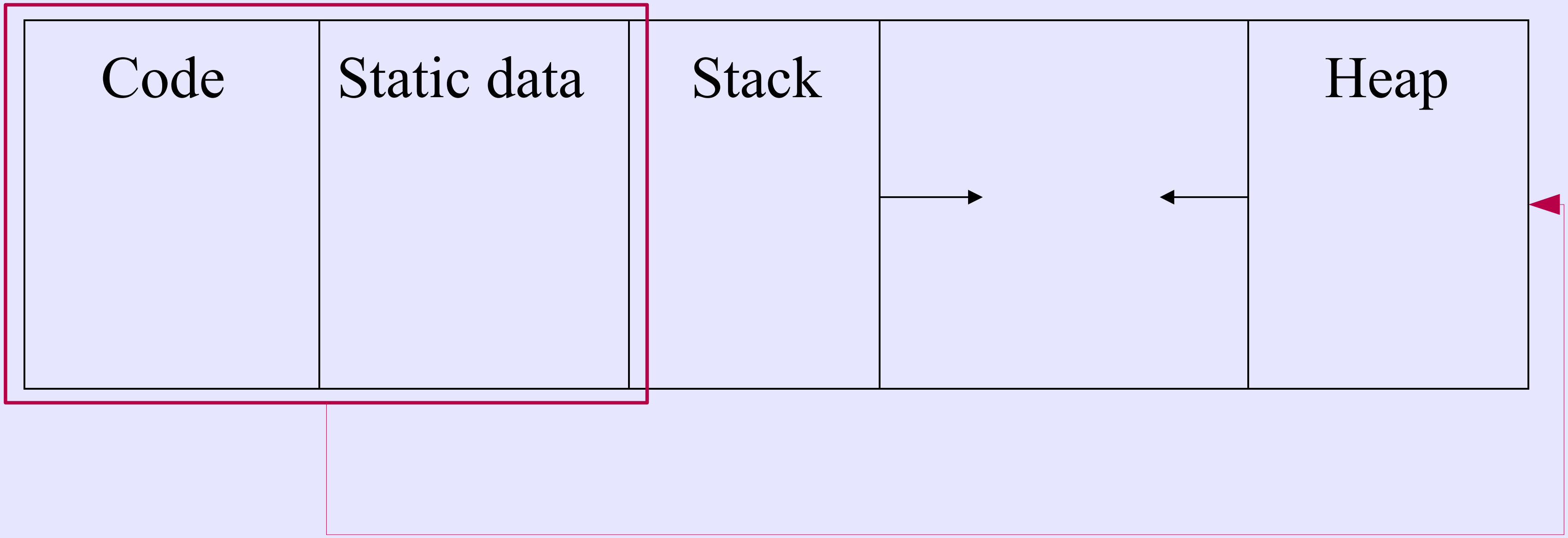
```
                 /* size    bytes */
typedef struct {
  int     f1;    /* 4       0-3   */
  char    f2;    /* 1       4     */
  int     f3;    /* 4       8-11  */
  double  f4;    /* 8       16-23 */
  char    f5;    /* 1       24    */
} padded;
```

Heap Allocation

Dynamic allocation may be explicit or implicit in the language.

- How can we keep track of what areas are free?
- How can we prevent fragmentation?
- Garbage -- storage location that becomes unreachable
- Dangling reference -- pointer to a memory location that is free

Storage Organization



Garbage Collection

Garbage collection is the process of locating and reclaiming unused memory.

- Three major classes of garbage collectors: mark-scan, copying, reference count.
- A collector that requires the program to halt during the collection is a stop/start collector; else it is a concurrent collector.
- Garbage collection is a big deal in functional or logic languages which use a lot of dynamic data.
- Avoid if possible!

Tabuľka symbolov

- Jediný dôvod udržovať ju aj počas behu je ladenie (debugging).
- Má byť malá a kompaktná. Doporučuje sa informáciu premennej dĺžky (identifikátory, konštanty) udržovať v heape a v tabuľke symbolov len smerníky.
- Z dôvodov jednoduchosti je najvhodnejšie jednoduché hašovanie alebo lineárne vyhľadávanie.
- Ak jazyk umožňuje „nested scopes“ (vnorené bloky) musí to podporovať aj tabuľka symbolov.

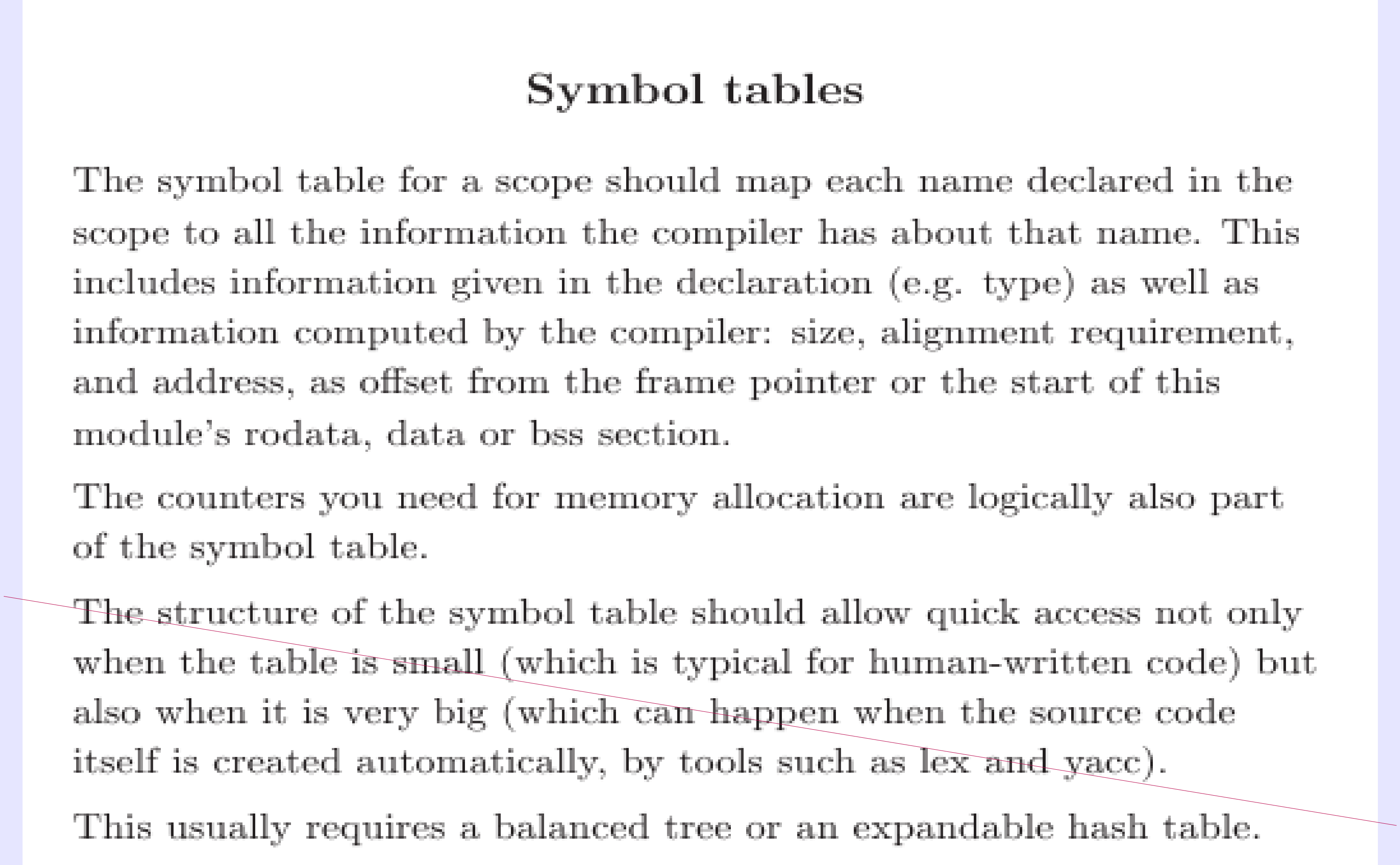
Symbol table structure

Each symbol table stores

- a pointer to the symbol table of the enclosing scope (if any)
- pointers to the symbol tables of the scopes it encloses
- whether it has its own stack frame (e.g. blocks don't)
- information about the named entities (such as types, variables, functions) declared in its scope.

The information associated with each name depends on what it is declared to be; you need to record different information for e.g. functions than for variables.

In C, only the global scope may contain function definitions. Other languages don't have this restriction.



Symbol tables

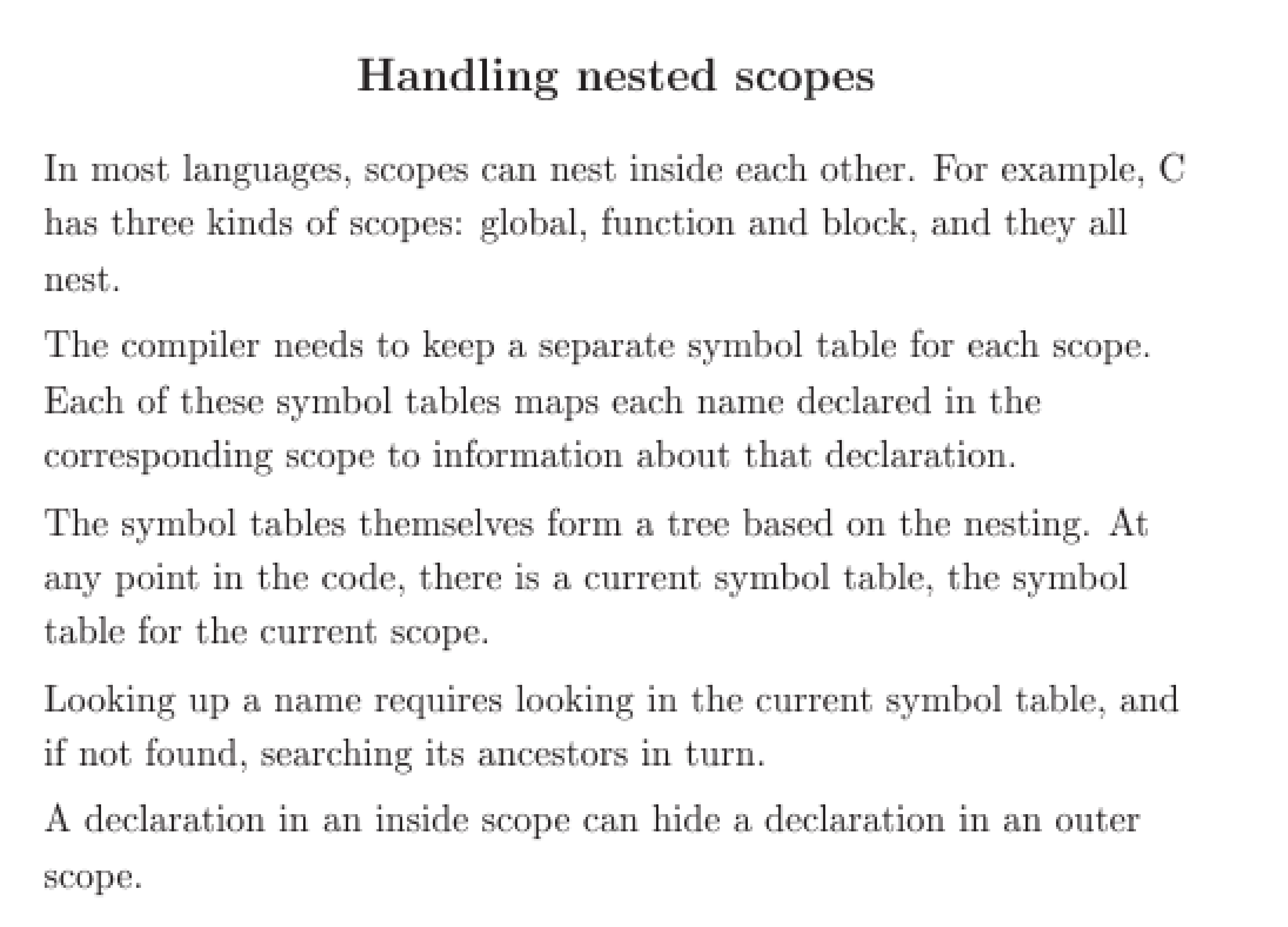
The symbol table for a scope should map each name declared in the scope to all the information the compiler has about that name. This includes information given in the declaration (e.g. type) as well as information computed by the compiler: size, alignment requirement, and address, as offset from the frame pointer or the start of this module's rodata, data or bss section.

The counters you need for memory allocation are logically also part of the symbol table.

The structure of the symbol table should allow quick access not only when the table is small (which is typical for human-written code) but also when it is very big (which can happen when the source code itself is created automatically, by tools such as lex and yacc).

This usually requires a balanced tree or an expandable hash table.

Preferoval by som separátnu kompiláciu. Program sa má skladať z relatívne malých nezávislých blokov.



Handling nested scopes

In most languages, scopes can nest inside each other. For example, C has three kinds of scopes: global, function and block, and they all nest.

The compiler needs to keep a separate symbol table for each scope. Each of these symbol tables maps each name declared in the corresponding scope to information about that declaration.

The symbol tables themselves form a tree based on the nesting. At any point in the code, there is a current symbol table, the symbol table for the current scope.

Looking up a name requires looking in the current symbol table, and if not found, searching its ancestors in turn.

A declaration in an inside scope can hide a declaration in an outer scope.

Hašovanie C (P. J. Weinberger)

```
#define PRIME 211
#define EOS '\0'
int hashpjw(s)
char *s;
{  char *p;
   unsigned h=0, g;
   for ( p = s; *p != EOS; p++ )
          { h = (h << 4) + (*p);
            if ( g < h&0xf0000000 ){ h = h ^ (g >> 24);
                                      h = h ^ g;
                                    }
          }
    return h % PRIME;
}
```