Languages and Compilers (SProg og Oversættere)

Run-time organization

Run-time organization

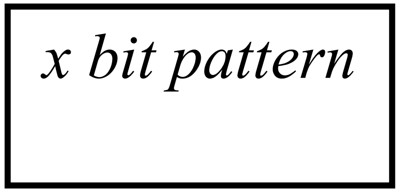
- Data representation (direct vs. indirect)
- Storage allocation strategies: static vs. stack dynamic
- Activation records (sometimes called frames)
- Routines and Parameter passing

Data Representation

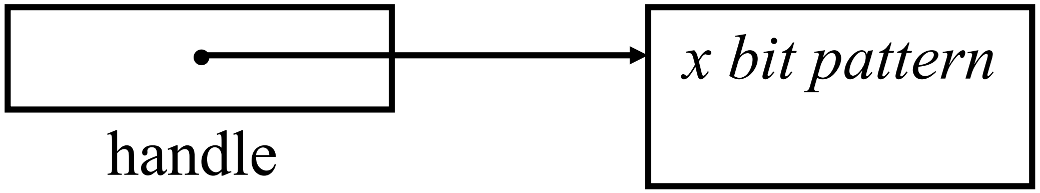
Important issues in data representation:

- **constant-size representation:** The representation of all values of a given type should occupy the same amount of space.
- **direct** versus **indirect** representation

Direct representation of a value *x*



Indirect representation of a value *x*

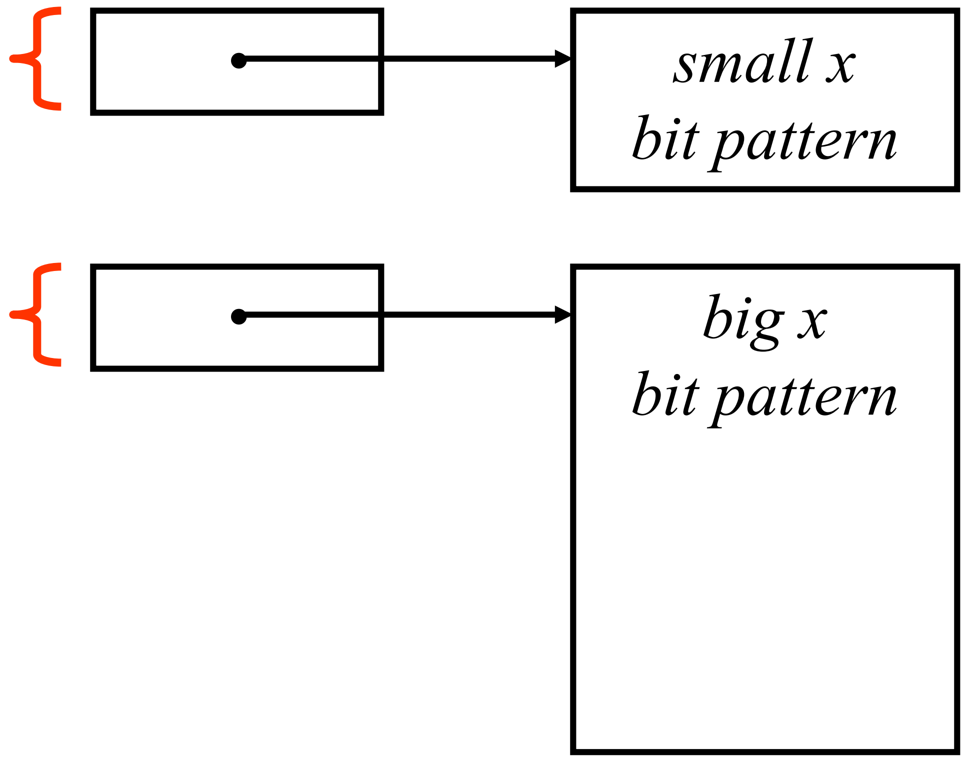


Indirect Representation

Q: What reasons could there be for choosing indirect representations?

To make the representation "constant size" even if representation requires different amounts of memory for different values.

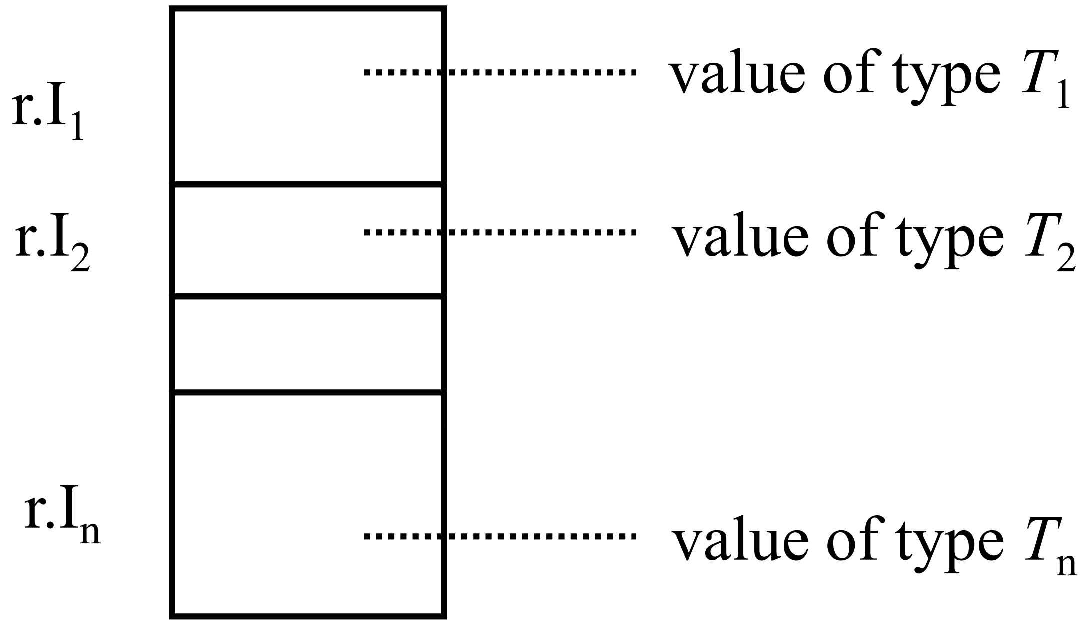
Both are represented by pointers =>Same size



Data Representation: Records

Records occur in some form or other in most programming languages:
Ada, Pascal, Triangle (here they are actually called records)
C, C++, C# (here they are called structs).

The usual representation of a record type is just the concatenation of individual representations of each of its component types.



Arrays

An array is a composite data type, an array value consists of multiple values of the same type. Arrays are in some sense like records, except that their elements all have the same type.

The elements of arrays are typically indexed using an integer value (In some languages such as for example Pascal, also other “ordinal” types can be used for indexing arrays).

Two kinds of arrays (with different runtime representation schemas):

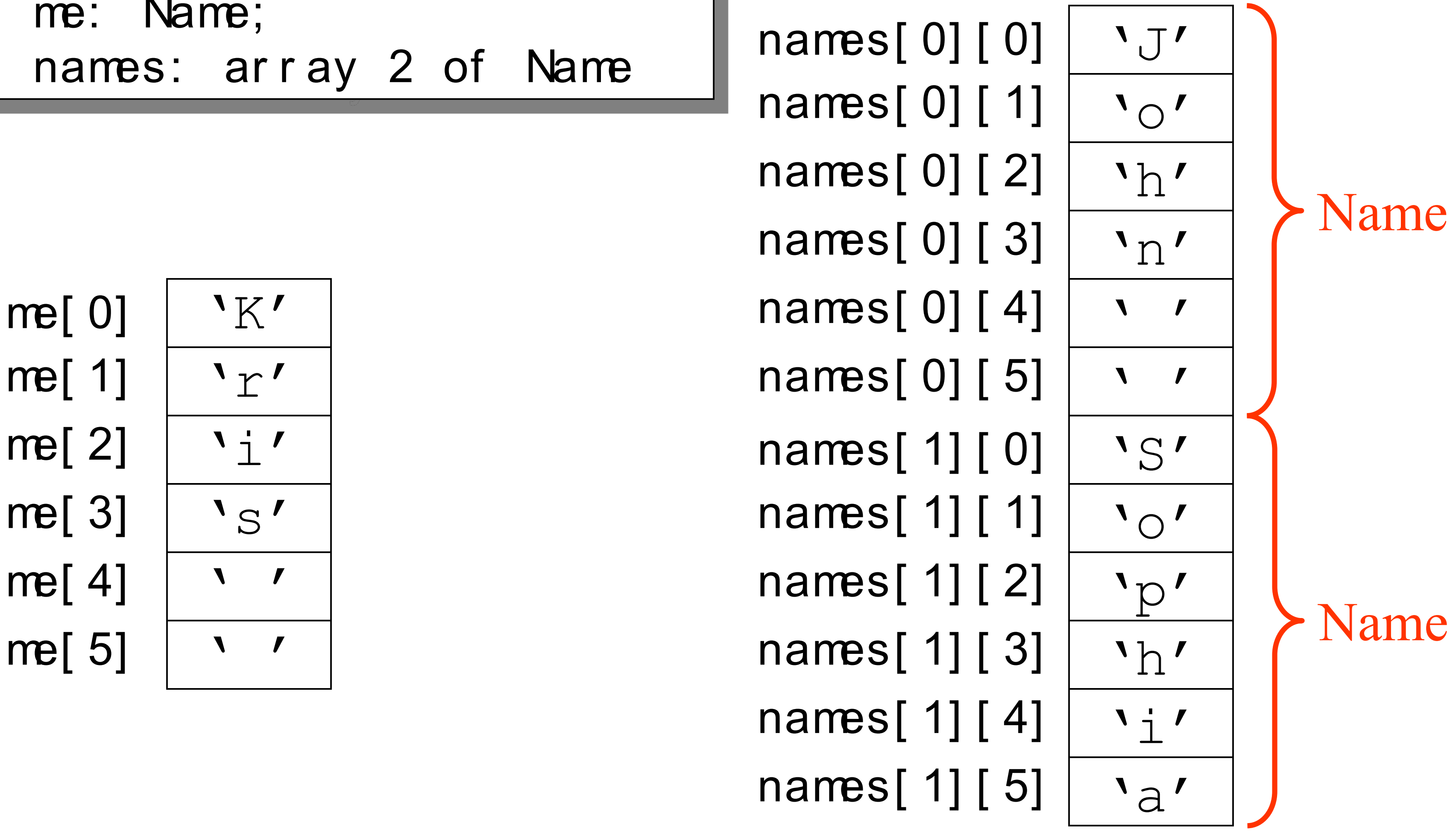
- **static** arrays: their **size** (number of elements) is **known** at **compile time**.
- **dynamic** arrays: their size can not be known at compile time because the number of elements may vary at run-time.

Q: Which are the “cheapest” arrays? Why?

Static Arrays

Example:

```
type Name = array 6 of Char;
var me: Name;
var names: array 2 of Name
```



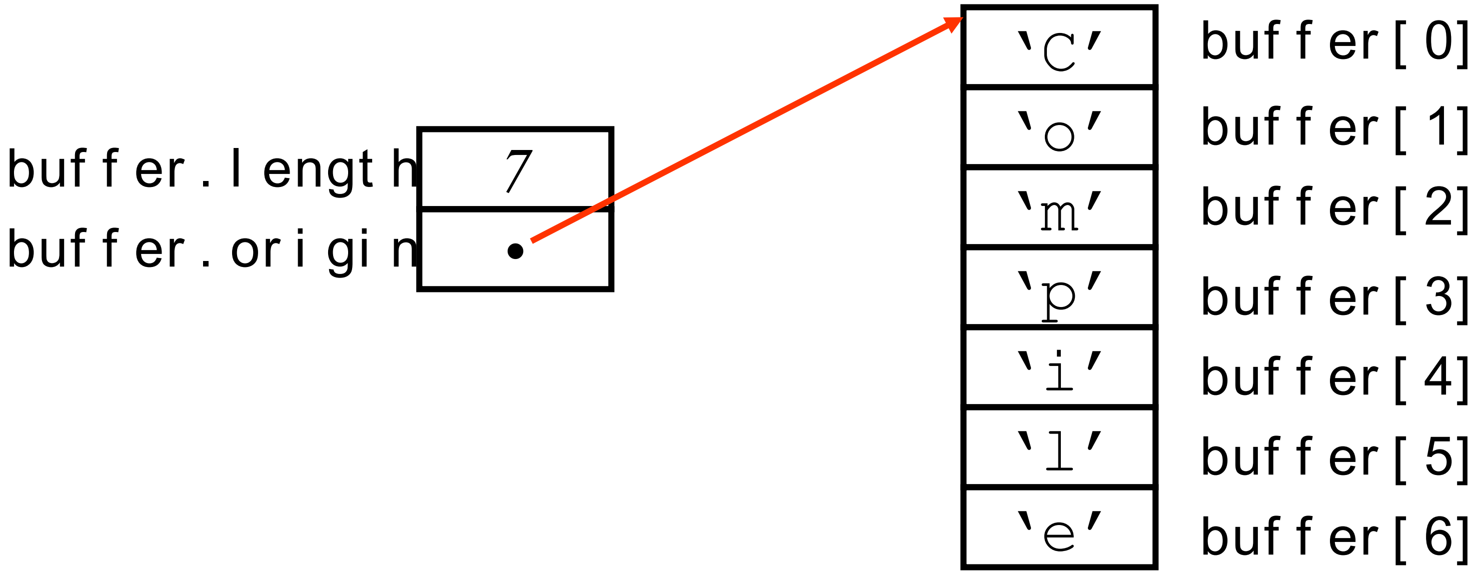
Dynamic Arrays

Java Arrays

```
char[ ] buffer;

buffer = new char[7];
```

A possible representation for Java arrays



Runtime Organization for OO Languages

Objects are a lot like records, and instance variables are a lot like fields.
=> The representation of objects is similar to that of a record.

Methods are a lot like procedures.
=> Implementation of methods is similar to routines.

But… there are differences:

Objects have methods as well as instance variables, records only have fields (except in C#).

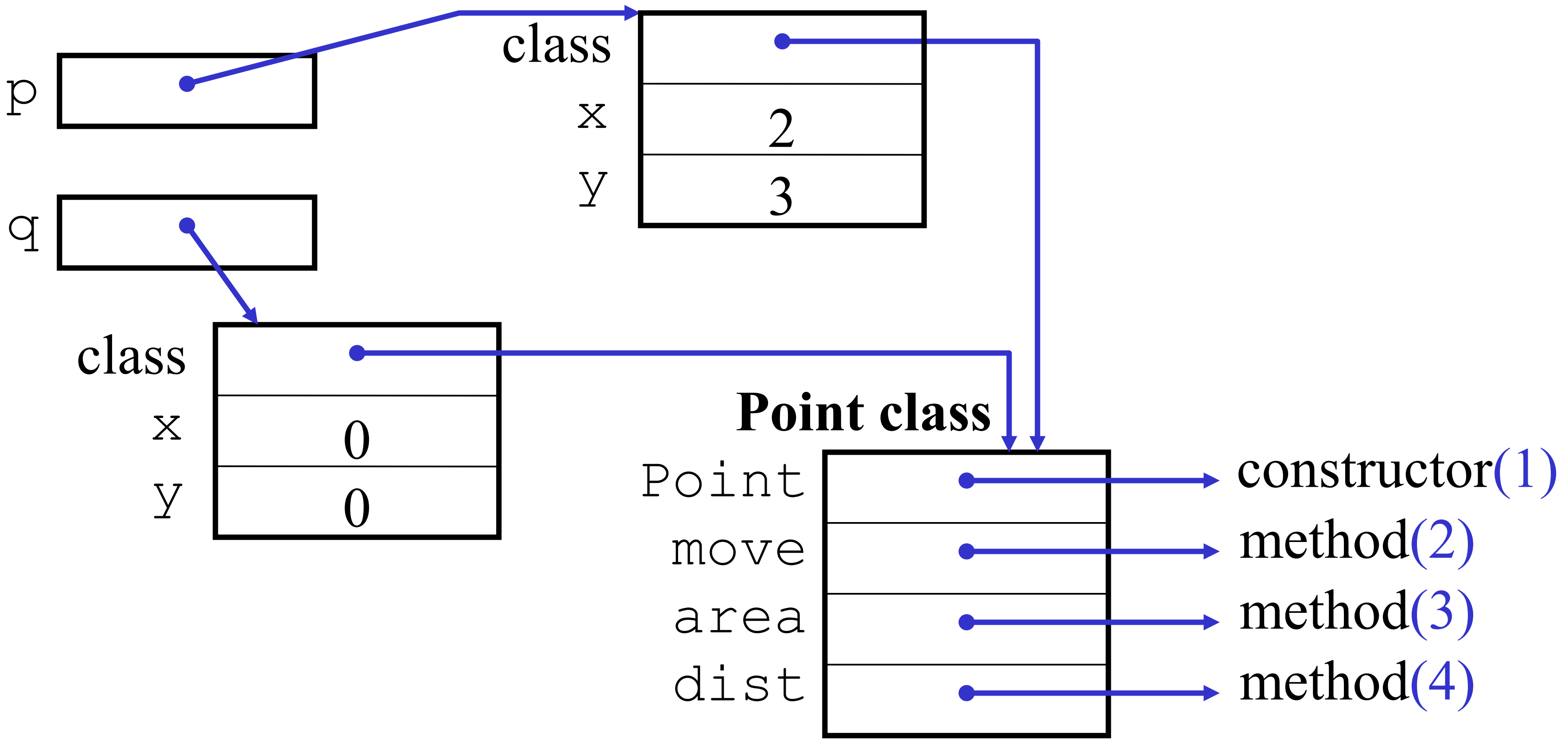
The methods have to somehow know what object they are associated with (so that methods can access the object's instance variables)

Example

Representation of a simple Java object (no inheritance)

```
Point p = new Point(2,3);
Point q = new Point(0,0);
```

new allocates an object in the heap



Where to put data?

Now we have looked at how program structures are implemented in a computer memory

Next we look at where to put them

We will cover 3 methods:

1) static allocation,

2) stack allocation, and

3) heap allocation.

Static Allocation

Originally, all data were global. Correspondingly, all memory allocation was static.

During compilation, data was simply placed at a fixed memory address for the entire execution of a program. This is called static allocation.

Examples are all assembly languages, Cobol, and Fortran.

Note: code is usually allocated statically

Stack Storage Allocation

Now we will look at allocation of local variables

Example: When do the variables in this program "exist"

```
    void Y() {
        int d;
        ... e;
         ... ; }
    void Z() {
        int f;
        ...; Y(); ... }
int main(){
int[3] a;
  bool b;
  char c;
    ..., Y(); ...; Z(); }
```

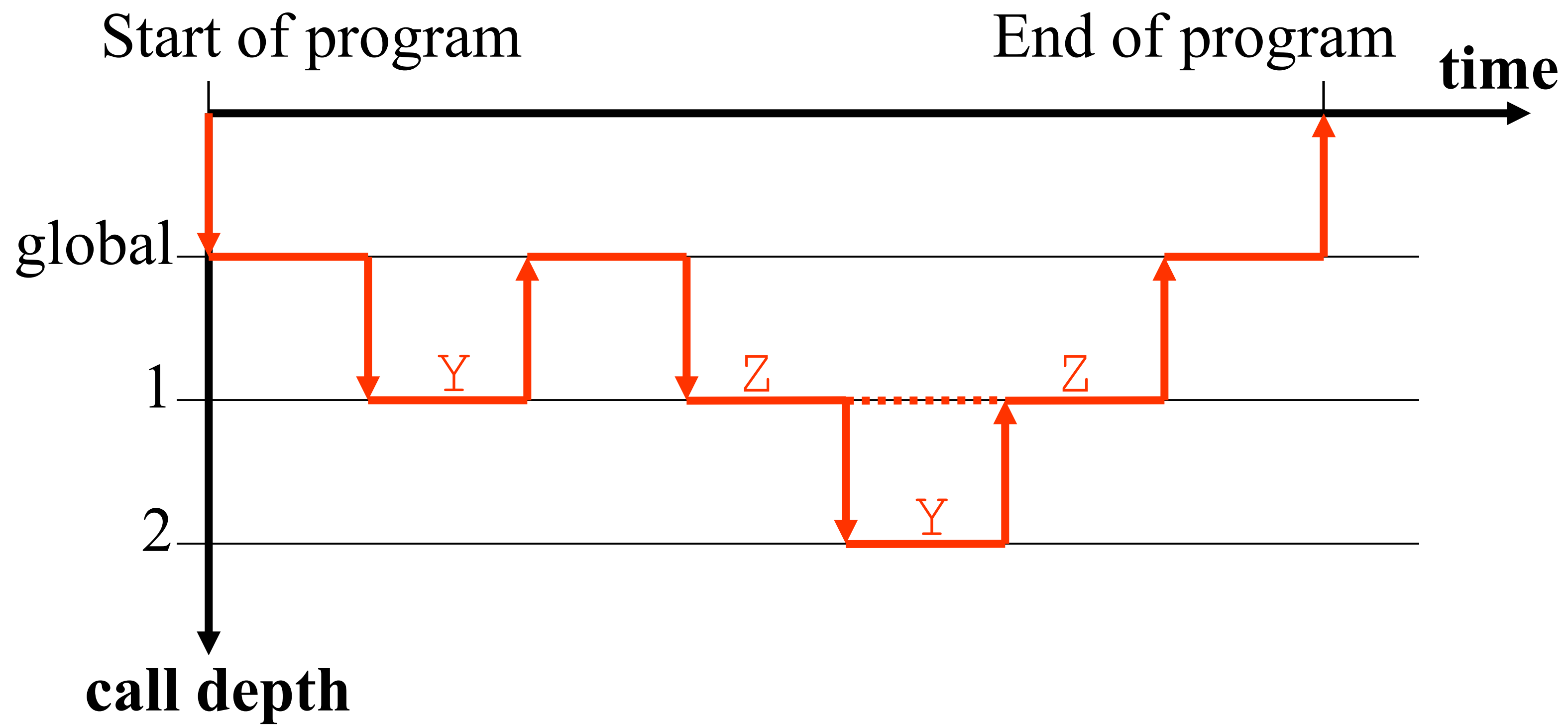
when procedure Y is active

when procedure Z is active

as long as the program is running

Stack Storage Allocation

A "picture" of our program running:



1) Procedure activation behaves like a stack (LIFO).
2) The local variables "live" as long as the procedure they are declared in.
1+2 => Allocation of locals on the "call stack" is a good model.

Recursion

```
int fact (int n) {
    if (n>1) return n* fact (n-1);
    else return 1;
}
```

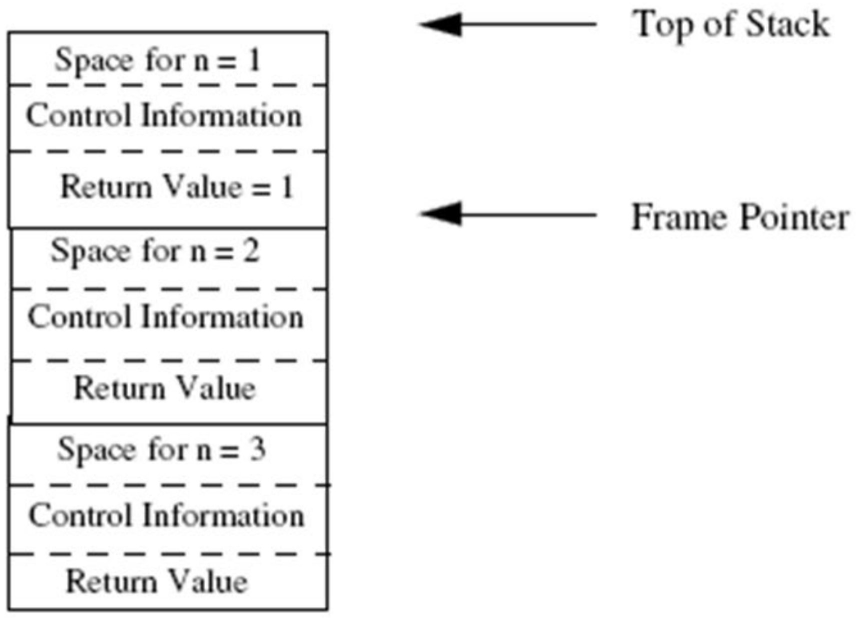
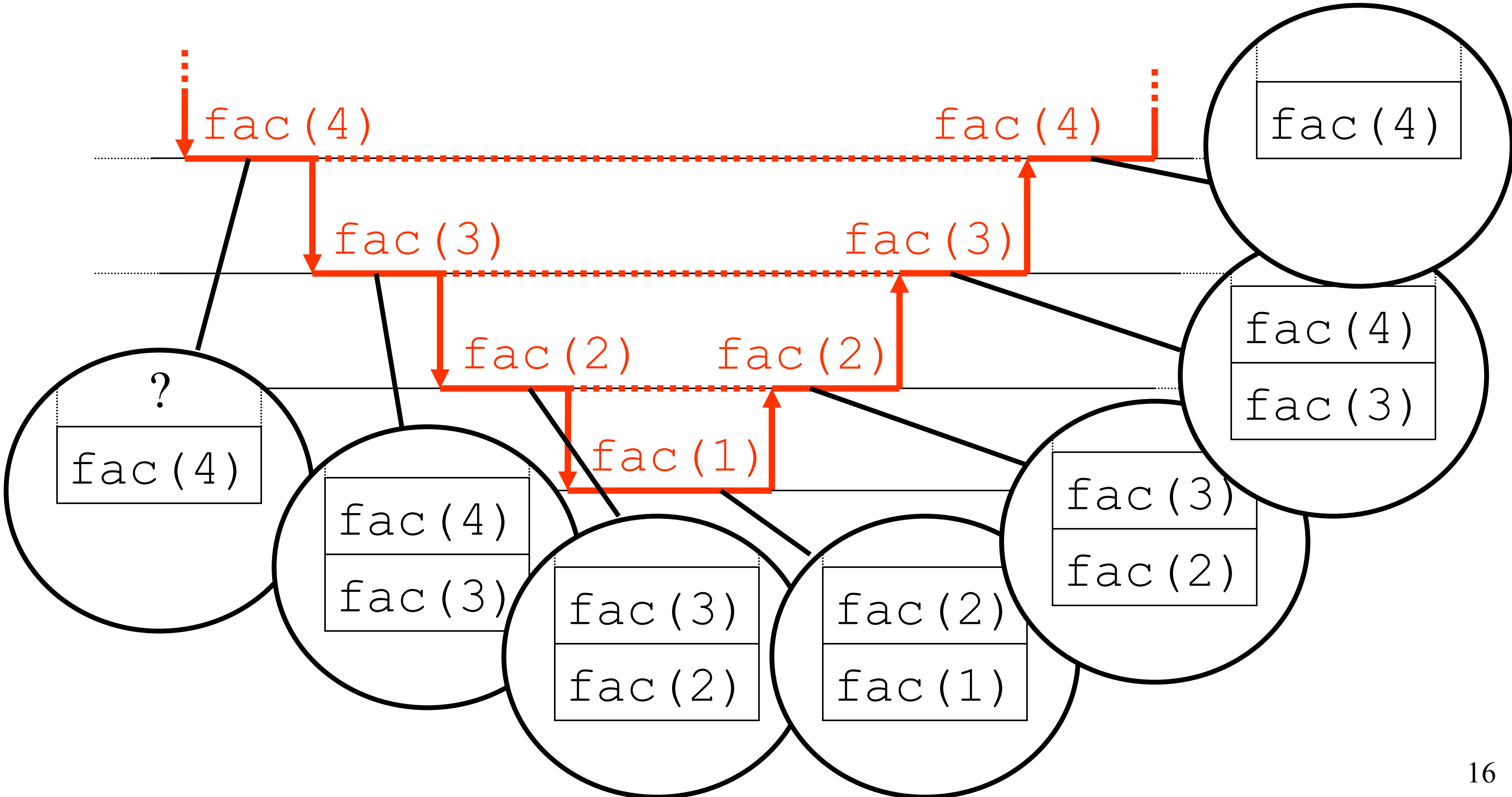


Figure 12.3: Runtime Stack for a Call of `fact(3)`

Recursion: General Idea

Why the stack allocation model works for recursion:
Like other function/procedure calls, lifetimes of local variables and parameters for recursive calls behave like a stack.



Nested functions/procedures

```
int p (int a) {
    int q (int b) { if (b <0) q (-b)  else return a+b; }
    return q (-10);
}
```

Methods cannot nest in C, Java, but in languages like Pascal, ML and Python they can. How to keep track of static block structure as above?

A static link points to the frame of the method that statically encloses the current method. (Fig. 12.6)

An alternative to using static links to access frames of enclosing methods is the use of a display. Here, we maintain a set of registers which comprise the display. (see Fig. 12,7)

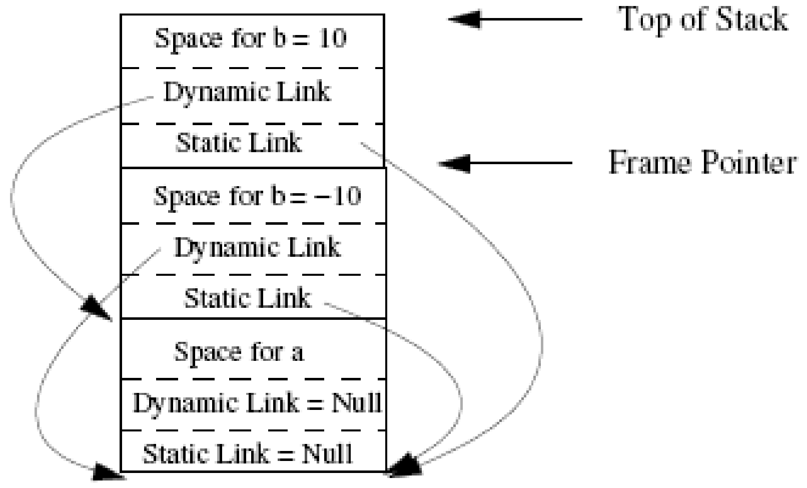


Figure 12.6: An Example of Static Links

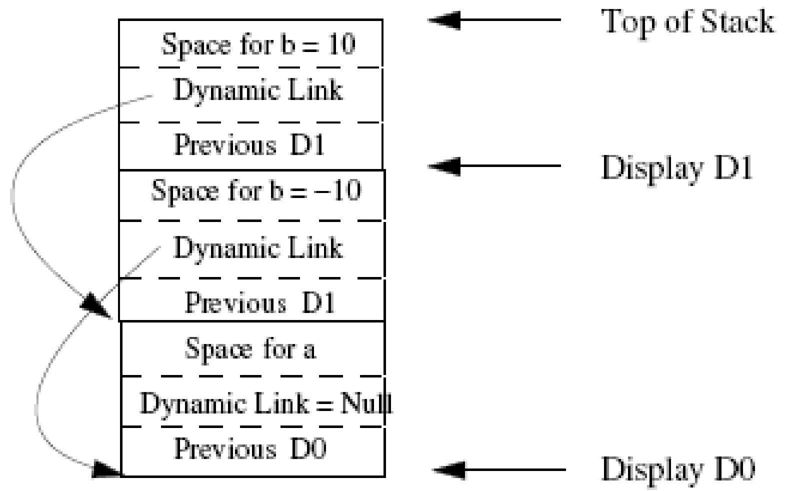


Figure 12.7: An Example of Display Registers

Routine Protocol Examples

Example 1: A possible routine protocol for a stack machine

- Passing of arguments:
  pass arguments on the top of the stack.
- Passing of return value:
  leave the return value on the stack top, in place of the arguments.

Note: this protocol puts no boundary on the number of arguments and the size of the arguments.

Most micro-processors, have registers as well as a stack. Such "mixed" machines also often use a protocol like this one.

Routine Protocol Examples

The routine protocol depends on the machine architecture (e.g. stack machine versus register machine).

Example 2: A possible routine protocol for a RM

- Passing of arguments:
  first argument in R1, second argument in R2, etc.
- Passing of return value:
  return the result (if any) in R0

Note: this example is simplistic:

- What if more arguments than registers?
- What if the representation of an argument is larger than can be stored in a register.

For RM protocols, the protocol usually also specifies who (caller or callee) is responsible for saving contents of registers.