Chapter 12 Runtime Support

Wuu Yang

National Chiao-Tung University, Taiwan, R.O.C.

first draft: June 11, 2010

current version: June 16, 2022

Chapter outline: Runtime support

1. Static allocation
2. Stack allocation
3. Arrays
4. Heap management
5. Region-based memory management

References

1. COMPILER, ASSEMBLER, LINKER AND LOADER: A BRIEF STORY, http://www.tenouk.com/ModuleW.html

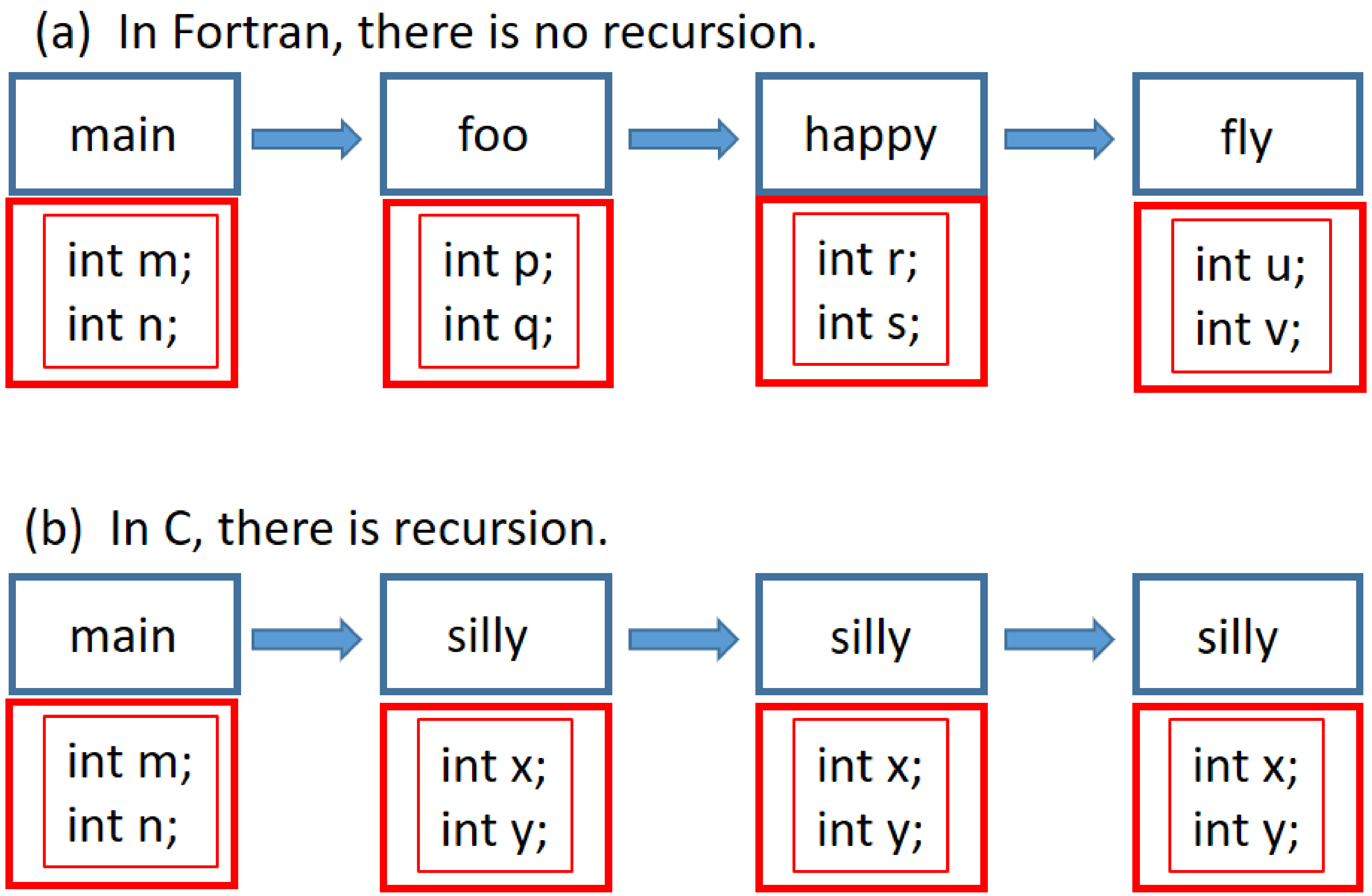


Figure 1: Where are the variables?

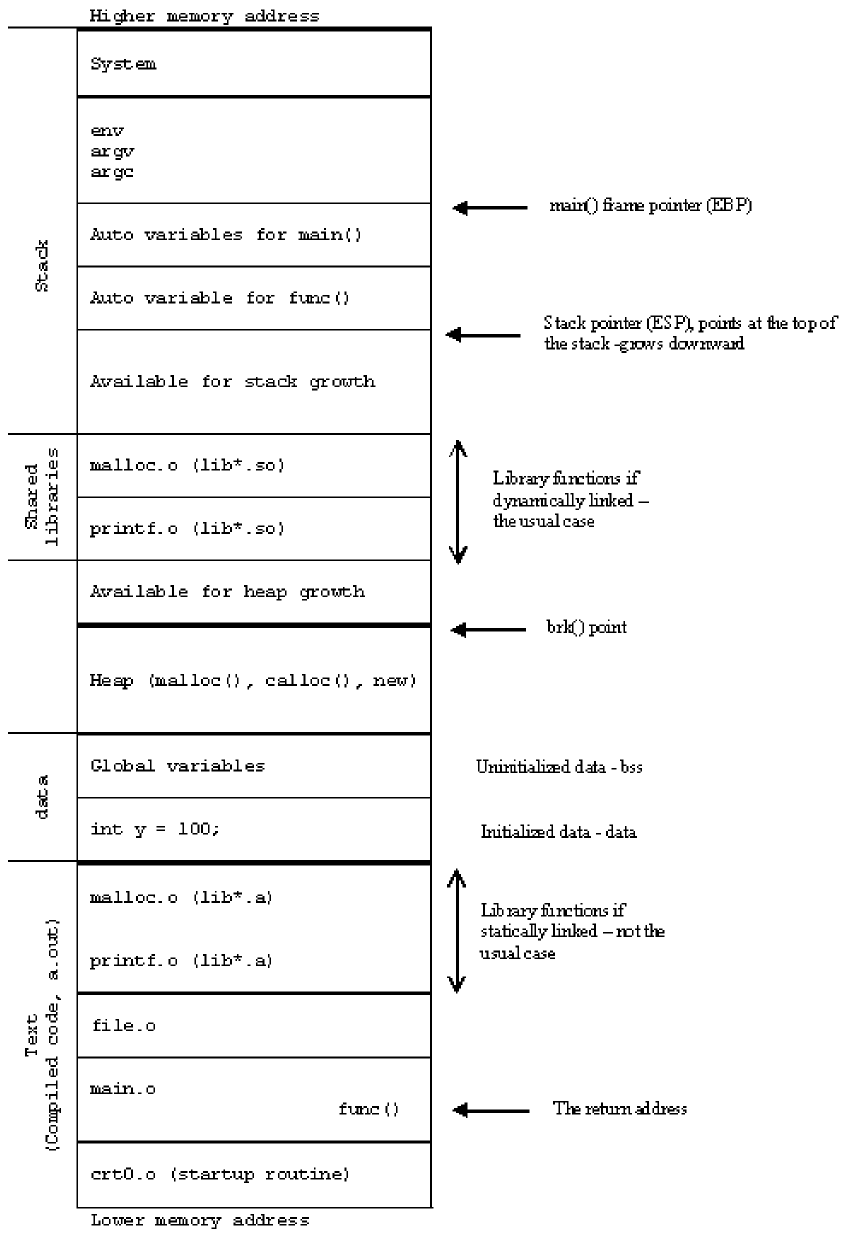

Higher memory address
Stack
System
env
argv
argc
Auto variables for main()
main() frame pointer (EBP)
Auto variable for func()
Stack pointer (ESP), points at the top of the stack -grows downward
Available for stack growth
Shared libraries
malloc.o (lib*.so)
printf.o (lib*.so)
Library functions if dynamically linked – the usual case
Available for heap growth
brk() point
Heap (malloc(), calloc(), new)
data
Global variables
Uninitialized data - bss
int y = 100;
Initialized data - data
Text (Compiled code, a.out)
malloc.o (lib*.a)
printf.o (lib*.a)
Library functions if statically linked – not the usual case
file.o
main.o
func()
The return address
crt0.o (startup routine)
Lower memory address

Figure 2: C's process memory layout on an x86.

What if there is only a `main` procedure in a program?

```
    public static void main(...)  { var a, b, c; ...  }
```

Originally, all data were global, with a life time spanning the entire period of program execution. Consequently, all storage allocation is `static`. All locations are fixed at compilation time.

Lisp and Algol 60 introduced local variables for procedures and the procedures may be recursive.

```
   procedure aa(...)  { var x, y; ...bb(...)...  }
   procedure bb(...)  { var u, v; ...aa(...)...  }
```

This leads to stack allocation. When a procedure is called, its frame is pushed onto a run-time stack. When a procedure returns, its frame, which must be on the top of stack, is popped off. A frame contains a procedure's local variables and some run-time information.

Lisp and many other modern programming languages allow dynamic allocation. Dynamic storage is allocated in the heap with `new` or `malloc`.

Try this program:

```
// gcc staticint.c -o staticint
#include <stdio.h>
int count;
void abc(int x) {
  static int si = 4;
  int li = 333;
  count ++;
  printf("count = %d. si = %d. li = %d.\n", count, si, li);
  si ++; li ++;
  if (count < 9) { abc(0); }
}

void main() {
  count = 3;
  abc(0);
}
```

Run-Time Storage Organization

Typical variable usages in a program:

```
1.    static int x;
2.    int abc(int y) {
3.        float z;
4.        static int w;
5.        . . .
6.        malloc(30);
7.        . . .
8.        abc(w) . . .;
9.        . . .
10.    }
11.    main → abc(...) → abc(...) → abc(...)
```

Where should all the variables and constants be placed? Remember procedures may be recursive. This means there could be multiple copies of a local variable.

1. `static var x`
2. constant 30
3. parameter `y`
4. return value (sometimes implicit)
5. local variable `z`
6. dynamic allocation `malloc()`
7. static local variable `w`

There can be exactly one copy for each global variable and `static` variable.

Constants may be duplicated. However, duplication is a waste and probably slows down program execution due to page faults and lost optimization opportunities.

There may be several copies for the arguments and the local variables of a procedure due to recursive calls.

Some storage is dynamically allocated at run time.

Note that some variables, such as the return value of a function, are implicitly created.

We need to worry about the *life span* of a data object: when it is created and when it is last used.

Storage allocation is closely related to linkers and loaders. (You may want to study the PE format for a deeper understanding of storage.)

Logically, the run-time storage of a program is a sequence of blank bytes. This empty space can be partitioned into three categories:

1. **static allocation**: as in Fortran.
2. **stack allocation**: for recursive functions.
3. **heap allocation**: for dynamic allocation.

Note that the cost of stack management is cheaper than that of heap management. Static storage does not cause any run-time cost.

§12.1 Static allocation

In Fortran, COBOL, and assembly languages. All data objects are allocated (by the compiler) at fixed locations for the entire duration of program execution. There is no run-time overhead. The bad side of static allocation is potentially wasting space.

Static allocation was easy initially. Later, *overlay* was introduced (as `common blocks` in Fortran) to save storage. But this is potentially dangerous.

In static allocation, the memory address of a data object is determined at compile time. There is no run-time overhead in computing the addresses. Static allocation is good for global variables, constants (i.e., literals), `static` and `extern` (in C) variables, static fields in C# and Java classes (even though classes are dynamically loaded), and program code.

There are two possibilities:

1. There is exactly one static area: Every static variable is bound to a fixed absolute address.
2. There are multiple static areas: Every static variable is bound to a pair:

   (StaticDataArea, offset)

   For instance, JVM could provide a static area especially for static fields of dynamically loaded classes. A compiler would not know the exact size of such a static area. If the starting address of a static area is not known at compilation time, a compiler could load that address into a register (a *global pointer*) and uses a pair

   (global pointer, offset)

   to address a data object when the static area is bound to virtual memory.

For separately compiled programs, binding is delayed until link time.

Static allocation is the most efficient. At run time, a program does not need to calculate the addresses of data objects.

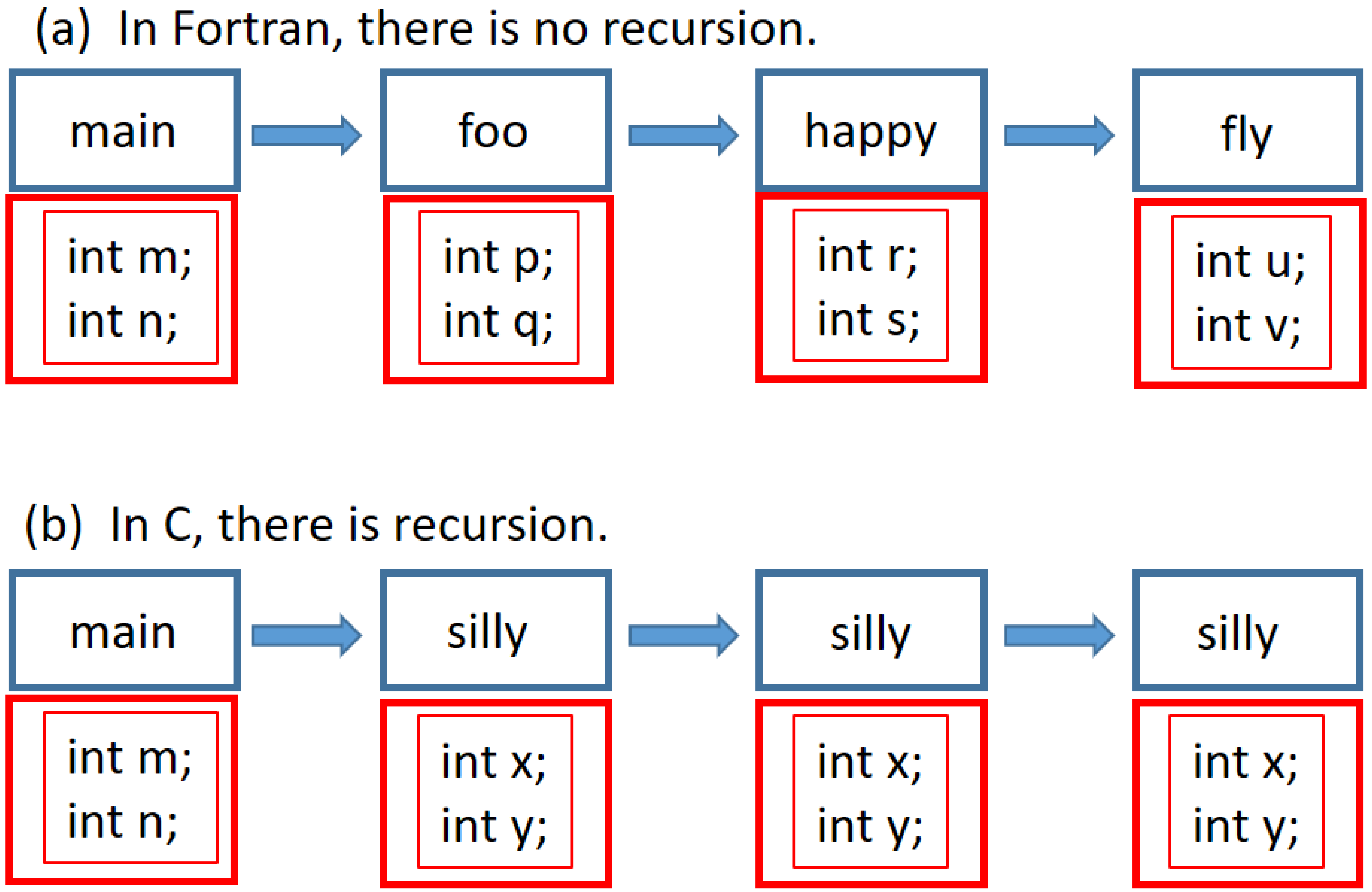


Figure 3: Where are the variables?

§12.2 Stack allocation

For (recursive) function invocations, each invocation requires a new copy of the function's local variables and parameters. The variables that are created last are freed first. Hence, we can use stack allocation. Stack allocation incurs run-time overhead in two aspects:

- When a procedure is called and returns, we need to adjust the *frame pointer* (which points to the *current frame*) and the *stack pointer* (which points to the free space).
- When a local variable is referenced, we need to calculate its address by adding an offset and the frame pointer.

Stack allocation is good for parameters, return values, and local variables of a procedure.

The stack contains *activation records* (ARs or *frames*). Different procedures have different frames. Push an AR when a procedure is called. Pop an AR when a procedure returns.

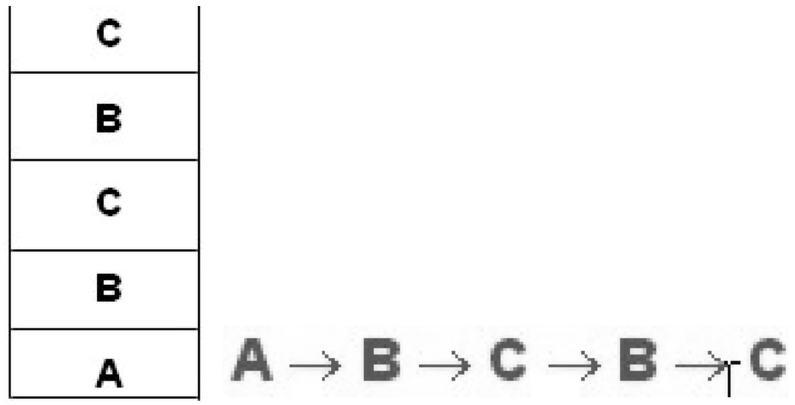


Figure 4: Run-time stack and the frames.

When a function is called, its activation record is pushed onto the run-time stack. AR contains function parameters, return values, local variables, and some pointers for organizing the stack. For example,

```
procedure P(a : integer)  is
      b : real;
      c : array (1..10) of real;
begin b := c(a) * 2.51; end
```

The AR looks like the following picture:

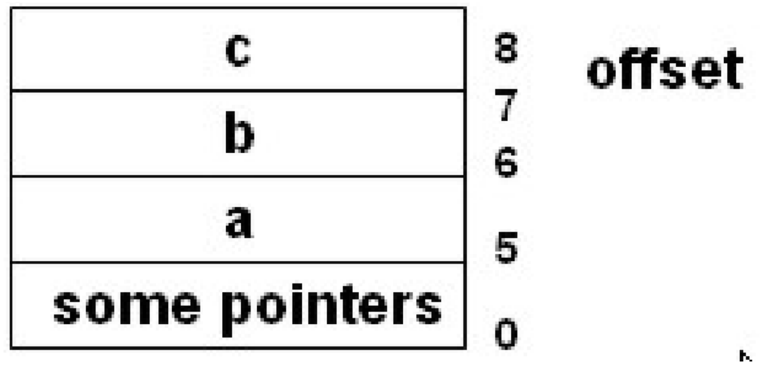


Figure 5: A frame.

Example. Sometimes, we need to insert padding between data objects due to architecture consideration. Thus, at all times, the top of the stack is properly aligned.

```
p(int a) {
   double b;
   double c[10];
   b = c[a] * 2.51;
}
```

Figure 12.1: A Simple Subprogram

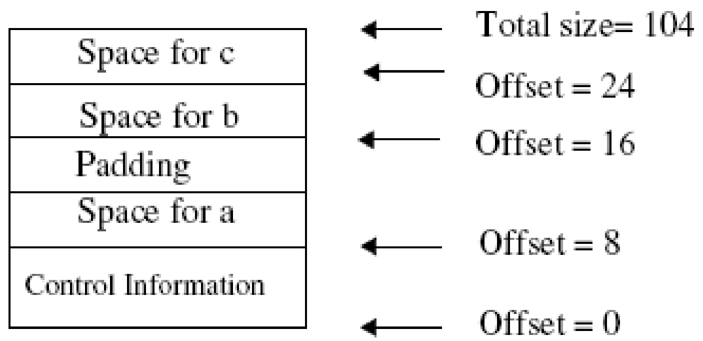


Figure 12.2: Frame for Procedure p

The actual address of a data object is the sum of its offset within the frame and the address of the current frame (i.e., the frame pointer).

At compile time, we may decide the offsets of the data objects within the frames. The offset information is stored in the identifier's entry in the symbol table. However, the address of the frame can only be determined at run time.

The compiler will calculate the offset of each data item and generate code to load the starting address of the frame. A data object is loaded with the pair

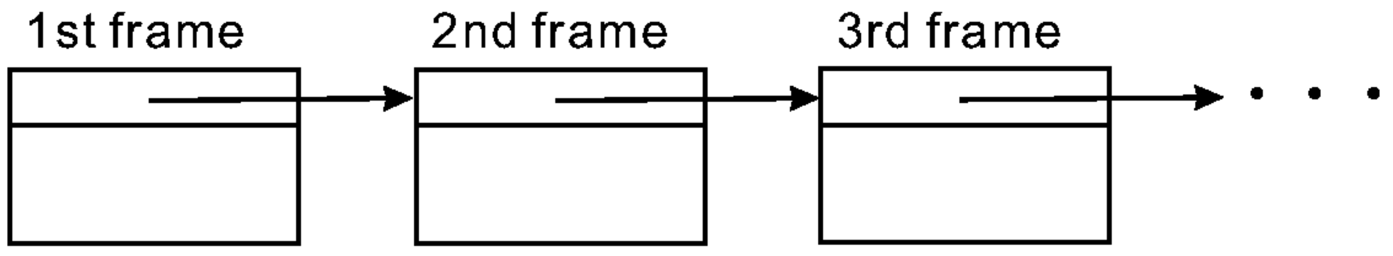
```
(register, offset)
```

(in one instruction when appropriate addressing modes are available).

The compiler may also determine the size of the frame of a procedure.

Question. Do we need to use a stack for ARs? How about a linked list (on the heap)?

An alternative allocation for frames:



Stack frame allocation as a linked list

A possible optimization is to avoid using the stack whenever possible. We may do a control flow analysis of the whole program to distinguish recursive and non-recursive procedures. For non-recursive procedures, we use static allocation. For recursive ones, we use the run-time stack.

§12.2.1 Field access in classes and structures

A field in a structure is addressed with a pair: the offset of the field within the structure and the address of the structure.

```
struct s {         // size is 96 bytes
  int a;           // offset = 0, pad 4 bytes for double.
  double b;        // offset = 8
  double c[10];    // offset = 16
}
struct s foo;
```

Assume `foo`'s address is 4000. Then `foo.b`'s address is 4008 (starting address + offset).

§12.2.2 Accessing frames at run time

There is a run-time stack during program execution. The stack contains frames, one for each invoked procedure. We need two pointers for the stack: The *frame pointer* points to the starting address of the frame on stack top. The *stack pointer* points to the free area above the topmost frame.

In theory, only one pointer is necessary *if* the size of the frame of every procedure is known at compile time. However, a frame might grow during program execution (for example, dynamic arrays). Thus, a compiler usually uses two pointers.

Example. There could be multiple invocations of a recursive procedure:

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

The run-time stack for the call `factorial(3)` is shown in Figure 12.3.

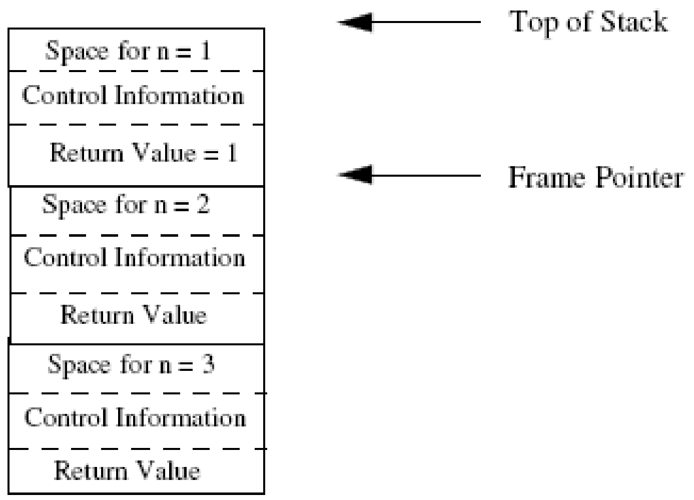


Figure 12.3: Runtime Stack for a Call of `fact(3)`

There is a slot in the frame for the return value. Many compilers put a scalar return value in a designated register instead. This saves a `store` and a `load` instructions.

Literals, such as 2.51, are stored in a static literal pool.

How about dynamic arrays?

```
procedure P(a : integer) is
     b : array[ 1 .. a ] of integer;
begin . . . b[3] . . . end  // reference an element

. . . P(9) . . .            // invoke procedure P
```

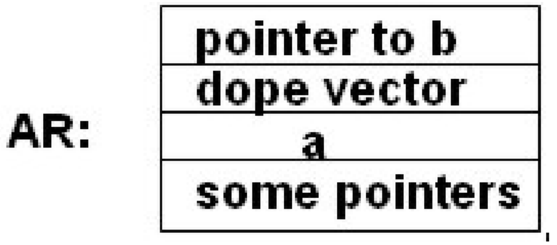
We do not know the size of the array at compile time.

Solution A dynamic array could be allocated on the heap during entry into a procedure.

After examining the life time of the dynamic array `b`, we have a better approach:

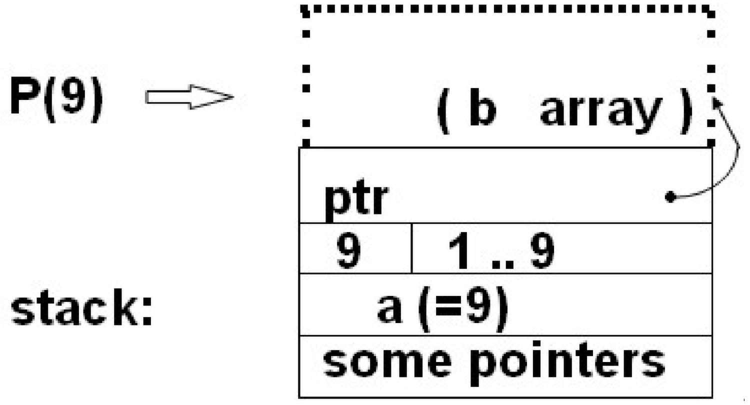
1. Each array type is represented by a dope vector (containing size and bounds).
2. Each dynamic array variable is represented by a pointer.

Both 1 and 2 are uninitialized and are on the stack.



At run time, when `P(9)` is called,

1. initialize the dope vector;
2. allocate space just on top of stack (or on the heap);
3. initialize the pointer.



The compiler needs to generate code for 1, 2, and 3.

A dynamic array is accessed indirectly through the pointer. This incurs run-time overhead. (We need to add offset and the frame pointer to obtain the address of the pointer and then add the pointer and element offset to obtain the address of the element. Finally access the element through that address.)

Other solutions?

Use dynamic patch, similar to block chaining in binary translators.

Example. The allocation of the following array is shown in Figure 12.10.

```
int examScore[numOfStudents()];
```

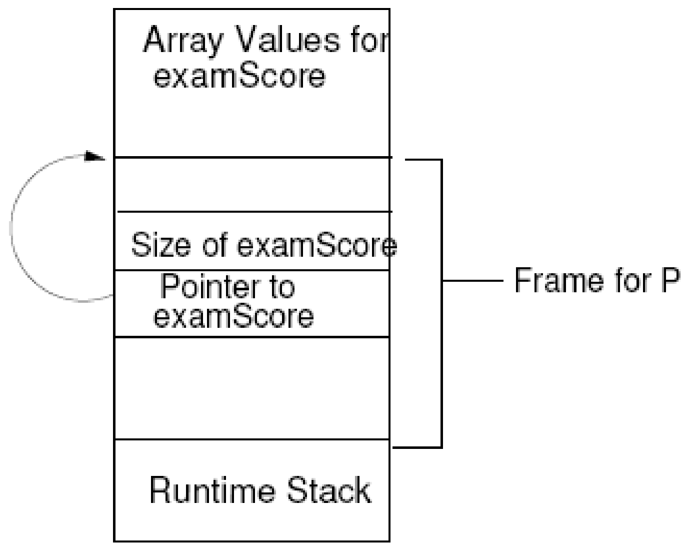


Figure 12.10: Allocation of a Dynamic Array

Java (the `Vector` class) supports *flexible arrays*, whose size can expand during execution. Flexible arrays are allocated on the heap. When an array needs expansion, a new block of memory is allocated and elements are copied from the old place to the new place.

When a dynamic or flexible array is passed as parameter, the location, size, and bounds are passed in order to locate the desired elements.

How about `static` variables in C?

```
int a;
void xyz( . . . ) {
   static int  b = 0;
   b = b + 1;  printf(b);
}
main() {
        xyz( . . . );
        xyz( . . . );
        xyz( . . . );
}  /* What values are printed?  */
```

Values of static variables are preserved across calls.

Solution Static variables are treated like global variables but their scopes are confined within individual procedures.

Subroutine calls are last-call-first-return. So we can use a stack. It is equally correct if we use heaps for frames during procedure calls. However, heap management incurs more run-time overhead.

Other features such as coroutines, interacting processes, shared variables, etc., are much more difficult and cannot be handled with a simple stack.

§12.2.4 Handling multiple scopes

(Accessing variables in multiple frames)

Many modern programming languages allow nested procedure declaration. Then a function may access variables in multiple scopes. Scopes may be nested in these languages.

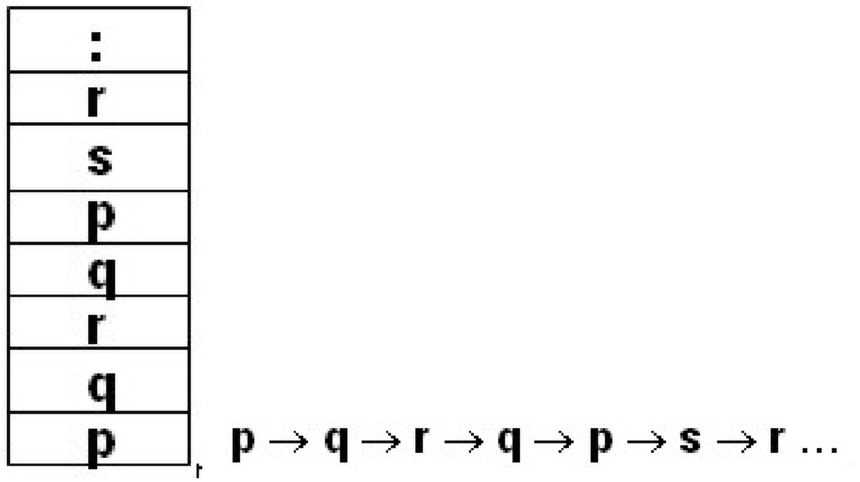
```
int p(int a) {
  int u
  int q(int b) {
    int v;
    return a + u + b + v;  // ok
  }
}
int r(int c) {
  c := a + b + c;  // error
}
```

Note the variables `a` and `u` are in the same scope (hence, in the same frame) while variables `b` and `c` are in a different frame.

Displays

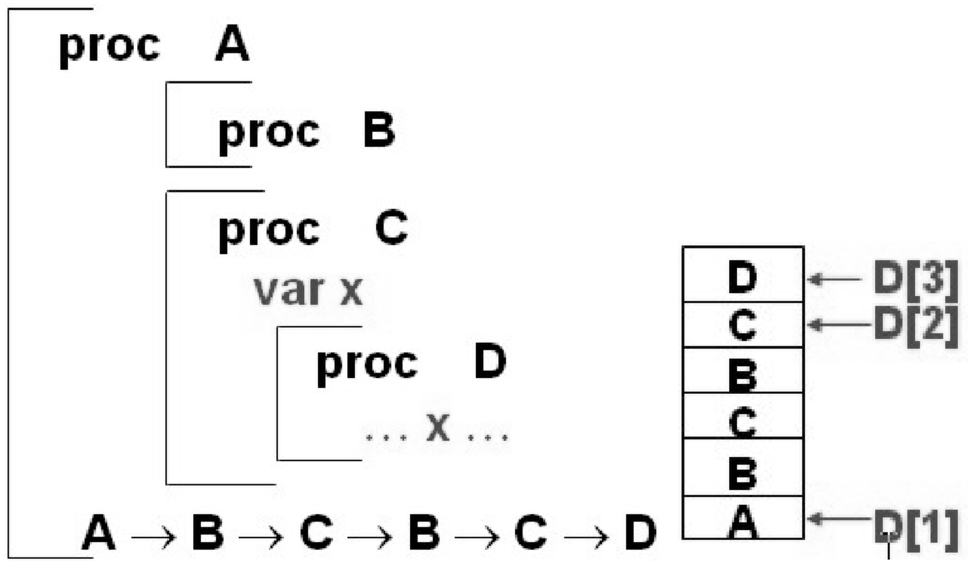
Displays are a set of pointers.

How to address the AR?



Need pointers to the ARs.

One register per AR? Not feasible because there could be thousands of ARs in the stack.



Note that at any instant, only the most recent ARs of the current procedure and the enclosing procedure are needed. Consider procedure `D` above. We do not need pointers to frames of B or old C.

This is called *static nesting*.

The stack can grow very big but the static nesting is quite shallow for ordinary programs.

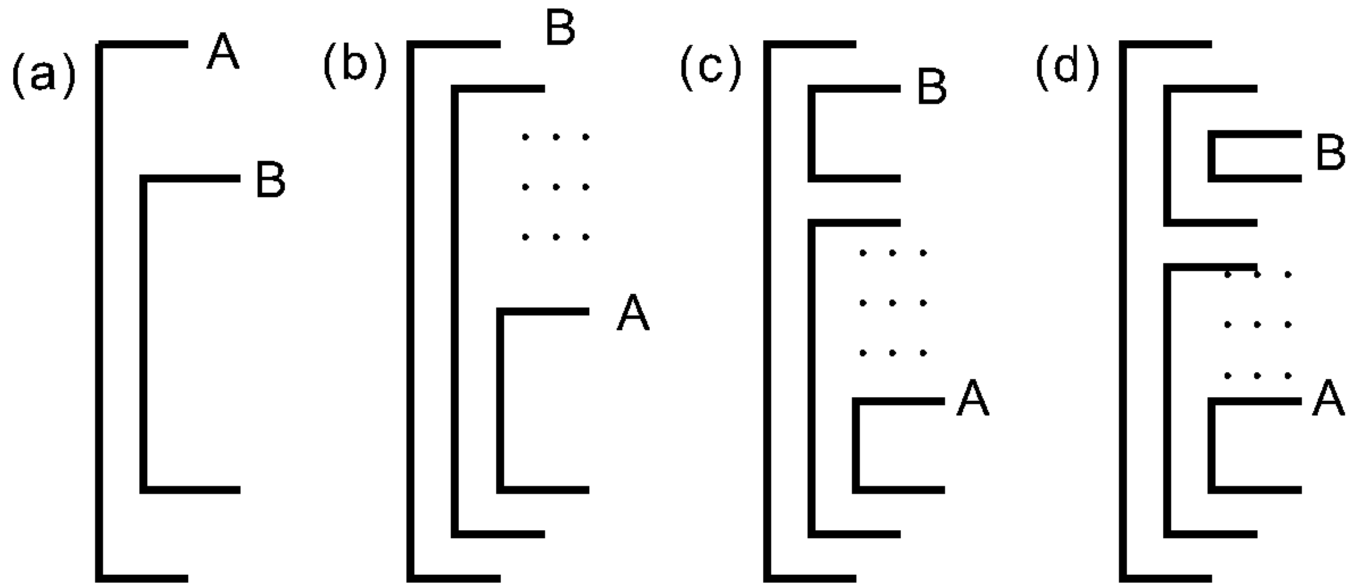
We can allocate one register for each nesting level. These registers are called *displays* (`D[1]`, `D[2]`, etc.).

See the previous slide.

The display can be stored in memory, rather than registers, if there are not enough registers in the CPU.

There are three cases when procedure A may call procedure B under the traditional scope rule:

1. A is the parent of B.
2. B is an ancestor of A.
3. B is a sibling of an ancestor of A. (B may be a sibling of A.)
4. In (d), A may not call B.



Upon procedure calls and returns, we need to adjust the display registers.

How?

1st approach: Save the entire display in the AR when a procedure is called. Restore display upon return.

This approach incurs much overhead.

2nd approach: Save only the display register for the level of the called procedure. Upon return, restore that register.

Note that when procedure `A` calls `B`, `B` must be one of (add a figure here)

- a procedure nested inside `A`
- a procedure that is a sibling of `A` or of an ancestor of `A`.

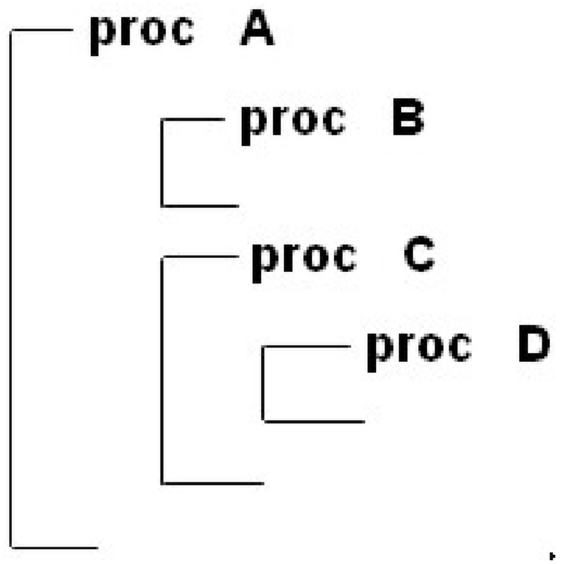
Who saves and restores?

Caller saves display registers in callee's AR. Callee restores it when the procedure returns.

Exercise. Suppose the calling sequence is

A → B → C → D → B → C.

Draw the displays.



Example. Display registers.

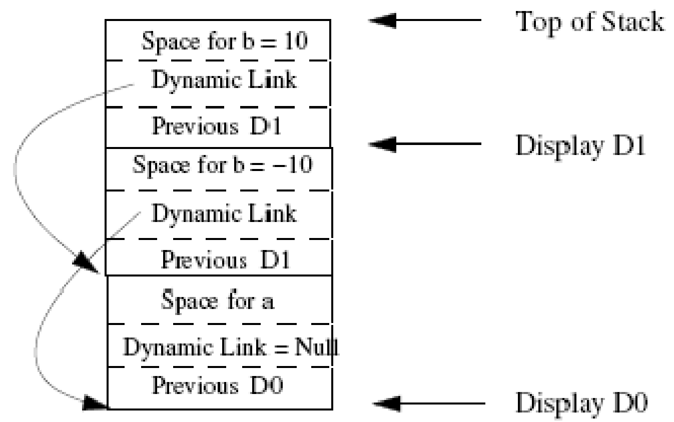


Figure 12.7: An Example of Display Registers

§12.2.5 Block-level vs. procedural-level ARs

In Ada, Algol 60, C, and Ada/CS, we may declare variables within blocks as well as within procedures.

```
Procedure  A  is
      a : integer;
begin
      begin b : integer;
            . . .
      end
      begin c : integer;
            . . .
      end
      . . .
end
```

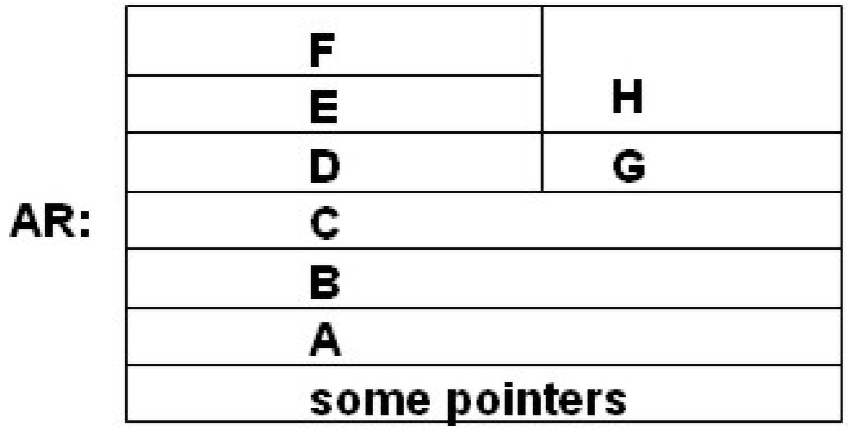
We may create a new AR for each block.

(-) more displays

(-) more run-time overhead

Or we may use procedural-level ARs. In this case, we may even overlay variables.

```
procedure  XYZ (A, B : integer) is
        C : integer
begin
        begin D, E : integer;
                begin F : integer;
                         . . .
                end
                . . .
        end
        begin G, H : integer;
                . . .
        end
        . . .
end
```


AR:
F
E
D
C
B
A
some pointers
H
G

Example. Figure 12.8 shows the procedure-level frame for the following code:

```
void p(int a) {
  int b;
  if (a > 0)
       { float c, d; ... }
  else { int e[10];  ... }
}
```

The procedure-level frame approach is called *scope flattening* since this reduces procedure nesting depths. This is effective especially when procedure-level register allocation is used.

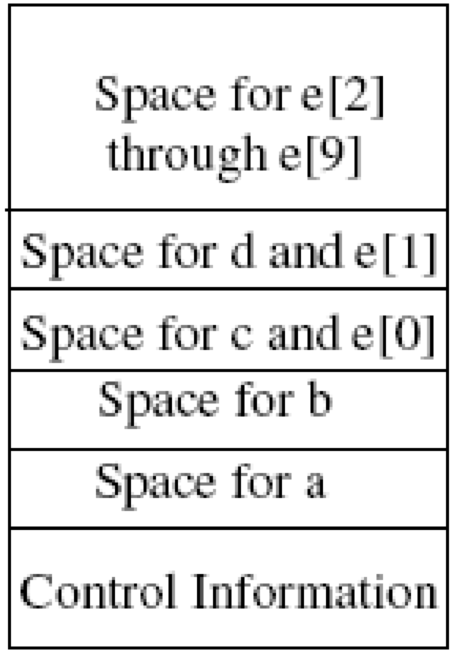
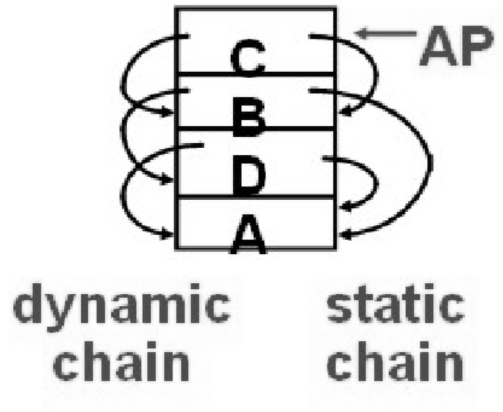


Figure 12.8: An Example of a Procedure-Level Frame

Static Chains and Dynamic Chains

```
proc   A
var  X;
    proc   B
    var  Y;
         proc   C
         begin ... X + Y ...
         end
     begin
        call C
     end
     proc   D
     begin
        call  B
     end
begin
   call  D
end
```

$A \rightarrow D \rightarrow B \rightarrow C$



For returning from a procedure, we need a chain of callers (*dynamic chain*).

For referencing variables, we need a chain of nested scopes (*static chain*).

Ex. Show how to reference `X` and `Y` in `C`.

We also need an AP (activation pointer or frame pointer, FP) that points to the topmost AR.

If we cannot use display registers, then we need to maintain the static chain.

static chain = display

To reference variables through static chain is slower than through display. But the performance is acceptable because
80% : local (via AP) or global data (fixed)
17% : immediately enclosing scope

Dynamic Chain

The run-time stack is maintained by two pointers: the frame pointer and the stack pointer.

To push a frame onto the run-time stack, we simply adjust the two pointers:

```
frame_pointer := stack_pointer;
stack_pointer := stack_pointer + size_of_callee_frame;
```

`size_of_callee_frame` is a constant computed by the compiler. When a procedure returns to the caller, we restore the two pointers:

```
stack_pointer := frame_pointer;
frame_pointer := frame_pointer - size_of_caller_frame;
```

However, it is not easy to identify the caller, not to mention the size of caller's frame.

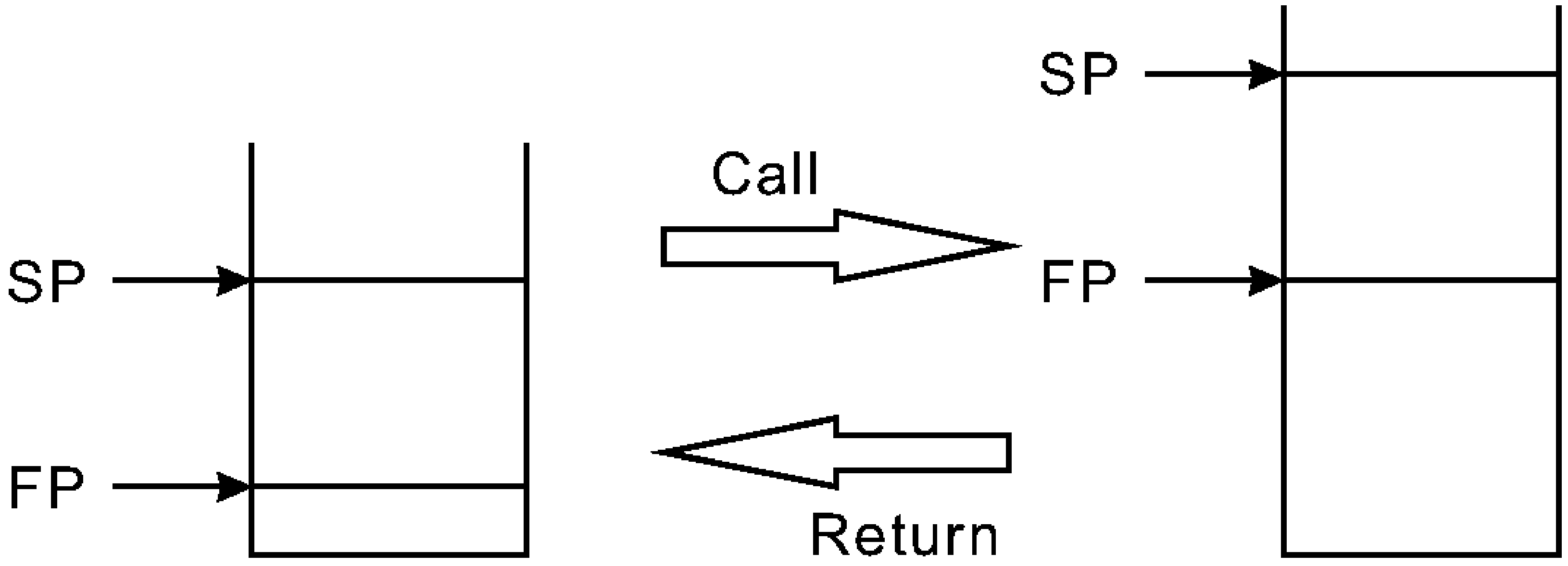


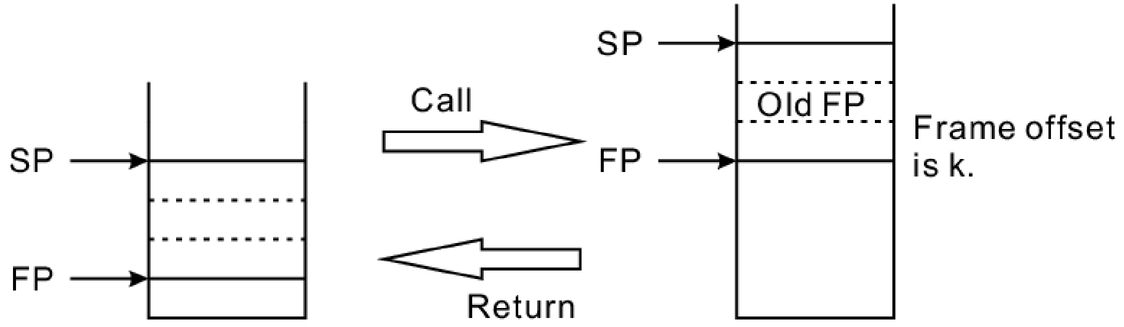
Figure 6: Manipulating frames on the run-time stack.

Therefore, we save the frame pointer (in the callee's frame) when a procedure is invoked. Here `k` is the offset within the frame for the saved frame pointer. `k` is the same for every procedure. (This means that the dynamic chain is stored in the `k`-th word in every frame.)

```
*(stack_pointer + k) := frame_pointer;
frame_pointer := stack_pointer;
stack_pointer := stack_pointer + size_of_callee_frame;
```

Then the frame pointer is restored when that procedure returns.

```
stack_pointer := frame_pointer;
frame_pointer := *(stack_pointer + k);
```



The saved framed pointers form a chain that links all the frames in the order the procedures are called.

Consider the following example:

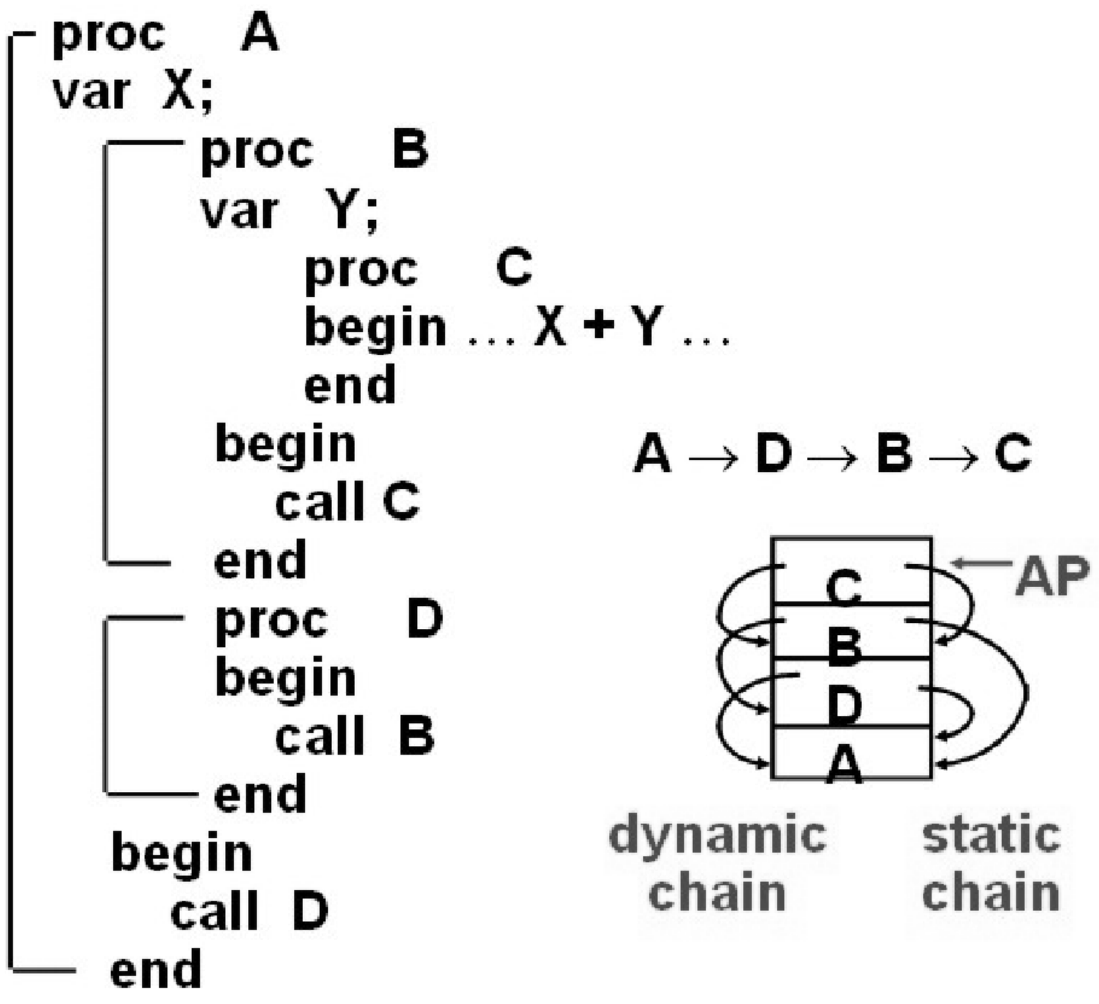
```
proc A
var x;
    proc B
    var y;
        proc C
        begin . . . x + y . . . end
    begin call C; end

    proc D
    begin call B; end

begin call D; end
```

The calling sequence is A $\rightarrow$ D $\rightarrow$ B $\rightarrow$ C . The run-time stack and the static and dynamic chains are shown below.

Static Chains and Dynamic Chains



```
proc   A
var  X;
    proc   B
    var  Y;
        proc   C
        begin ... X + Y ...
        end
    begin
        call C
    end
    proc   D
    begin
        call  B
    end
begin
   call  D
end
```

Figure 7: Static and dynamic chains.

Static Chain

On the other hand, to access non-local variables, we need to make use of the *static chain*. In the above example, procedure `C` references the names `x` and `y`. According to the usual scope rule, the name `x` refers to the variable `x` declared in procedure `A`. This means that `x` resides in `A`'s frame. The compiler can decide the offset of `x` in `A`'s frame. During execution, we still need to find `A`'s frame.

Similarly, the name `y` refers to the variable `y` declared in procedure `B`. Thus, we need to find `B`'s frame during execution.

Note that, when procedure `C` executes, we never need to know `D`'s frame since `C` can never reference any variables of `D`.

In summary, when a procedure, say `C`, executes, we need to know `C`'s frame as well as the frames of all enclosing procedures (that is, procedures `B` and `A`, but not `D`).

These frames of enclosing procedures are located through the *static chain*.

The frame of a procedure includes a pointer to the frame of the immediately enclosing procedure.

In the above example, the frames of procedures `B` and `D` include a pointer to `A`'s frame. Similarly, `C`'s frame includes a pointer to `B`'s frame.

These pointers form the static chain.

The static chain reflects the textual nesting structure of the program.

When a procedure, say `B`, is called, we need to establish a static link from `B`'s frame to the frame of `B`'s parent. In the above example, when `D` calls `B`, a static link from `B`'s frame to `A`'s frame is established. Similarly, when `B` calls `C`, a static link from `C`'s frame to `B`'s frame is established.

When a non-local variable `x` is referenced in a procedure `C`, the compiler can determine the procedure, say `A`, in which `x` is declared as well as the offset of `x` in `A`'s frame. The difference of the textual nesting levels of `C` and `A` is the number of static links that need to be traversed from `C`'s frame to `A`'s frame.

Consider the reference to `x` in procedure `C` in the above example. According to usual scope rule this `x` refers to the variable `x` declared in procedure `A`. The textual nesting levels of `A` and `C` are 1 and 3, respectively. Hence, the compiler generates code to traverse two static links to locate `A`'s frame and then access `x` through `x`'s offset in `A`'s frame.

Accessing non-local variables, thus, incurs more run-time overhead than accessing a local variable. Fortunately, only about 20% of variable accesses are non-local.

Example. Figure 12.4 shows the dynamic chain for Figure 12.3.

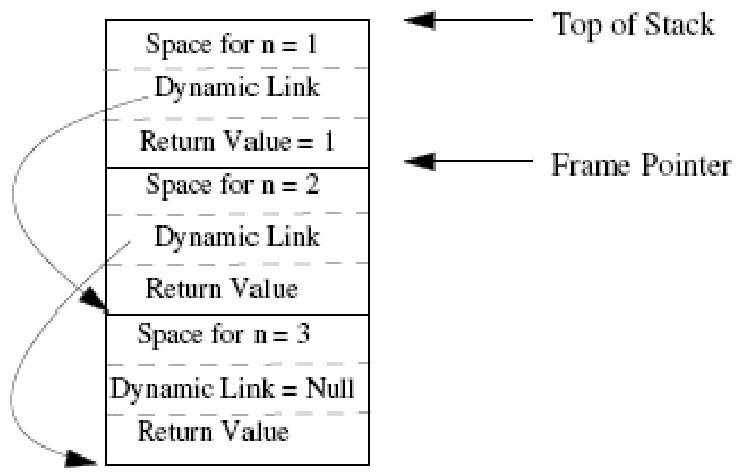


Figure 12.4: Runtime Stack for a Call of `fact(3)`with Dynamic Links

Example. Figure 12.6 shows the static chain for Figure 12.7.

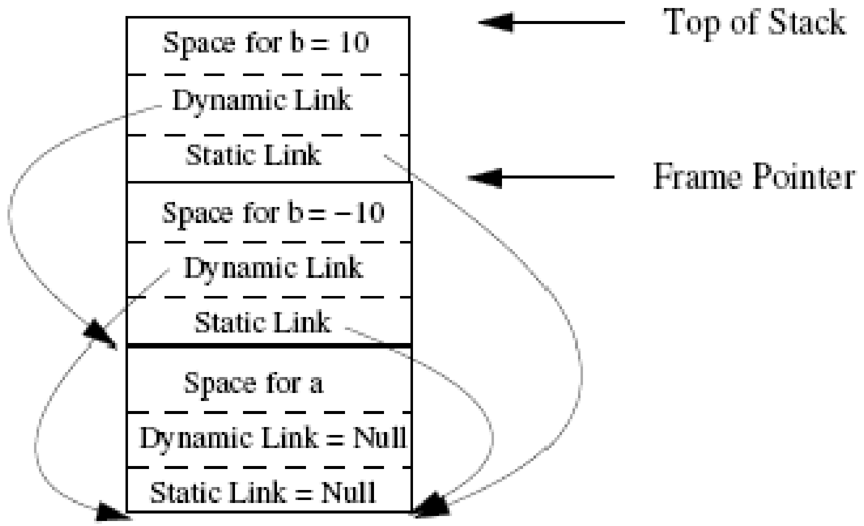


Figure 12.6: An Example of Static Links

For the C language, there is no static chain since procedures cannot be nested.

(from chapter 11) In Java, a method can reference three kinds of variables:

1. local variables (including parameters)
2. fields of an object (considered as global variables)
3. static variables of a class (considered as global variables)

Note that in Java as well as C, C++, and C#, procedures may not be nested. Variable usage is simpler.

```
class A {
  static int x;        // static variable
  float  y;            // field
  int foo(char z) {    // parameter
    double w;          // local variable
  }
}
```

There are only global and local variables. Global variables are statically allocated while local variables are allocated on the frame.

Example. (**Object reference as an implicit parameter**) Consider the following Java class:

```
class k {
  int a;          // field
  int sum() {
    int b = 42;
    return a + b;
  }
}
```

When

```
obj.sum()
```

is called, a reference to `obj` is passed as the 1st parameter. The run-time stack is shown in Figure 12.5:

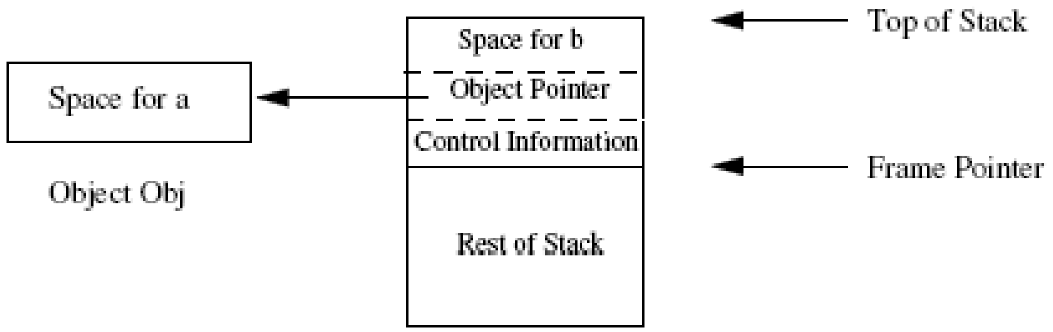


Figure 12.5: Accessing Local and Member data in Java

Suppose class B is a subclass of class A. All (static and instance) fields of A are copied to class B.

§1.2.2.6 More about frames

closures Usually, pointers point to data. C allows pointers to functions. C++ allows pointers to member functions.

When invoking a function f through a pointer, we need another pointer—which points to the *environment* of the particular instance of the function. This is because the invoked function f may reference variables defined in the enclosing scope. The two pointers together are called a *closure*. Closures are used in formal procedures.

In functional languages, such as Lisp, Scheme, and ML, functions are first-class. They may be stored in a variable, constructed during execution, passed as a parameter, and returned as a return value.

Runtime creation of functions is useful, as demonstrated by the following *memorizing* technique:

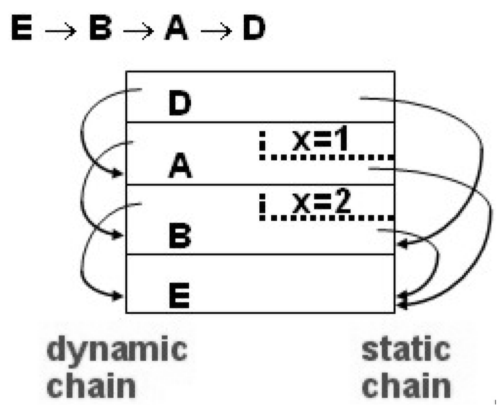
```
fun memo(fct, parm) = let val ans = fct(parm) in
  (ans, fn x => if x = parm then ans else fct(x))
end;
```

Formal Procedures

A procedure may be passed as a parameter and is later activated. Ex.

```
proc E( );
     proc A ( F: proc )
     var    x : integer;
     begin  x := 1; F( ); end

     proc B( );
     var    x : integer;
            proc  D( )
            begin write(x); end -- what is printed here?
     begin  x := 2; A( D ); end
begin B( ); end
```



If we follow the dynamic chain to find x, 1 is printed. This is called *dynamic binding*. In dynamic binding, we use the environment where the instance of `D` is activated.

If we follow the static chain to find x, 2 is printed. This is called *static binding*. In static binding, we use the environment where the instance of `D` is created.

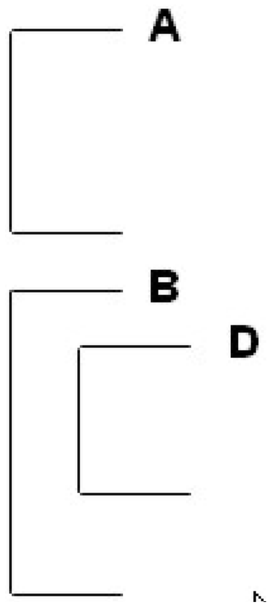
Only early LISP implementations use dynamic binding. Modern descendants of LISP use static binding. Algol 60 and its descendants all use static binding.

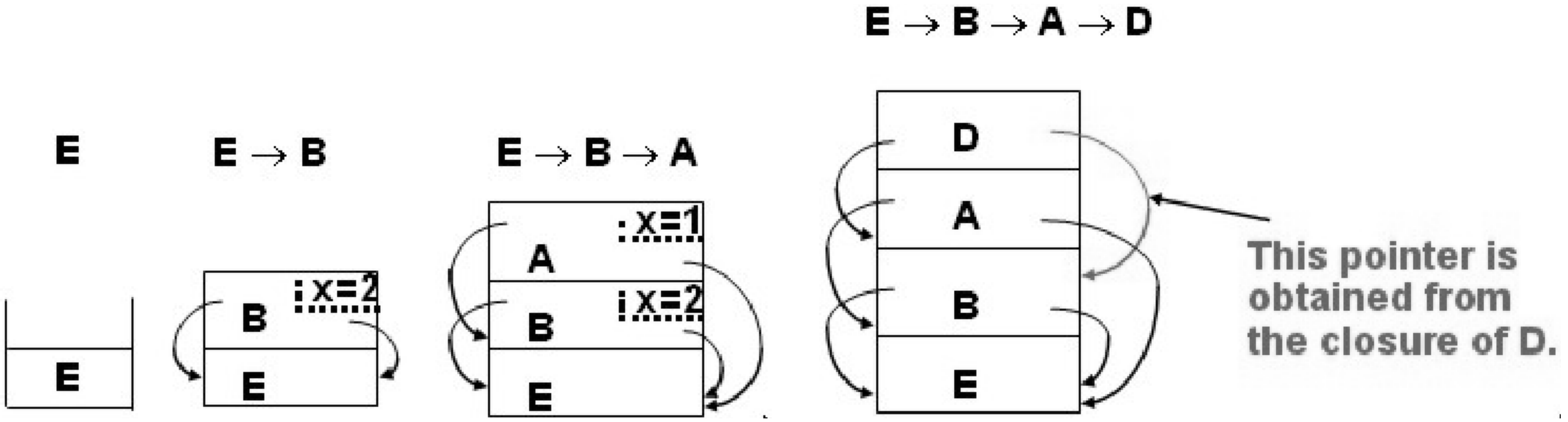
How to implement formal procedures in static binding?

When `B` passes `D` to `A`, we need to pass two things (together they are called a *closure*):

1. a pointer to D's code
2. a pointer to B's AR

Consider the following example:

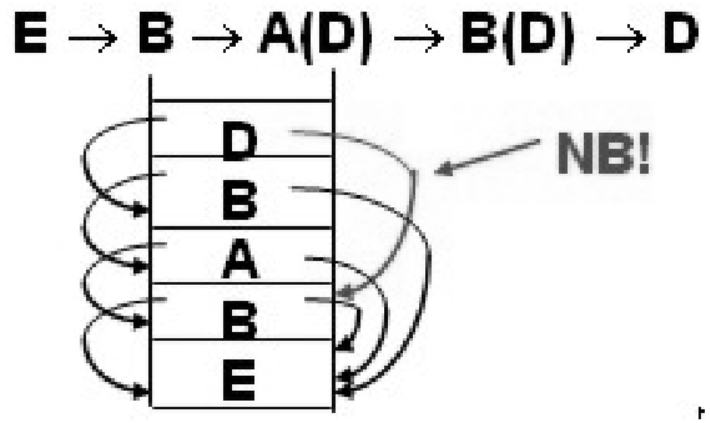




E
E → B
E → B → A
E → B → A → D
E
B
x=2
E
A
x=1
B
x=2
E
D
A
B
E
This pointer is obtained from the closure of D.

Example. Here is a more complicated example.

```
proc  E
    proc  A( F : proc )
    begin
        if … then B(F) else F( )
    end
    proc  B ( F: proc )
        proc  D( )
        begin
            …
        end
    begin
      if … then A(D) else F( )
    end
begin
    B( )
end
```



§9.6.2. Use Displays to Implement Formal Procedures

closure = code address + display registers

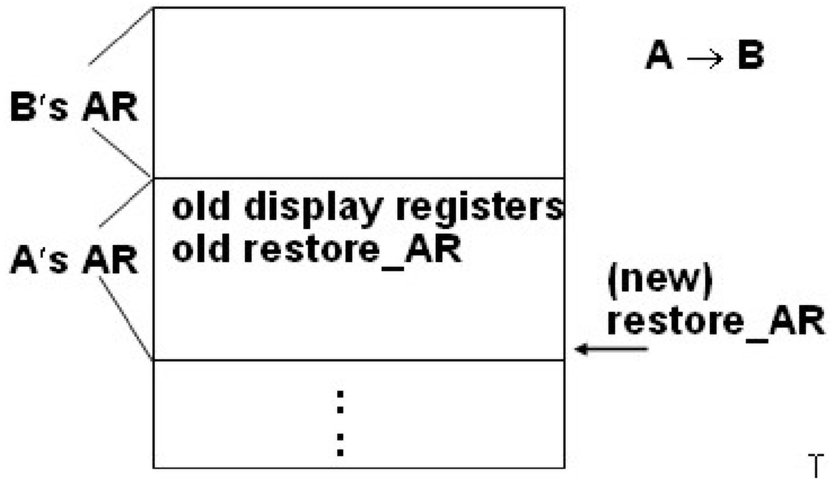
Need a pointer to the caller's AR. Call it `restore_AR`.

When procedure `A` calls formal procedure `D`,

1. Save `restore_AR` in `A`'s AR.
2. $\boxed{\text{Save display registers in A's AR.}}$
3. `restore_AR` := pointer to `A`'s AR.
4. $\boxed{\text{Load display registers from D's closure.}}$
5. Allocate `D`'s frame and adjust its dynamic chain.
6. Jump to `D`'s starting address.

When formal procedure `D` returns,

1. Free `D`'s frame.
2. $\boxed{\text{Use } \texttt{restore\_AR} \text{ to restore all display registers from } \texttt{A}\text{'s AR.}}$
3. Restore `restore_AR` from `A`'s AR.



This method incurs no extra cost for normal procedure calls.

However, it is difficult to handle non-local `goto`'s. Fortunately, non-local `goto`'s are rare.

Example. Function pointers in C:

```
void  dmi_decode(void) { . . . }

void  dmi_scan_machine(void) { dmi_table (dmi_decode); }

void  dmi_table( void (*decode)(void) ) { (*decode)(); }
      // decode is a parameter to function dmi_table
```

Perspective

Formal procedures are hard to understand.

Formal procedures in C are *flat*. C is a flat language. In C, a **closure = address of code** (in C).

Ada allows no formal procedures at all.

Formal procedures are useful in certain situations. (when?)

In the above discussion, procedures may be used as parameters, but not as return values. When procedures may serve as return values, the implementation would be more complex. For example, (in C-like notation)

```
int (*)() f(int x) {
  int g(int y) { return x + y; }
  return g;
}

int (*h)() = f(3);
int z = h(5);
```

The problem with this example is that we need to keep the frame of procedure `f` after it has returned. This will affect the structure of the run-time stack.

```
program Main(output);
   function f(level: int; function arg: int): int;
      function local: int;
      begin local := level; end;
   begin
      if level > 10     then f := f(level - 1, local)
      else if level > 1 then f := f(level - 1, arg)
      else                   f := arg;  /*actually invoke arg()*/
   end

   function dummy: int;
   begin /* do nothing */ end;

begin /* main */
   writeln("The answer is: ", f(13, dummy));
end. /*main */
```

```
program Main(Output);
    function f(Level : Integer; function Arg : Integer) : Integer;
        function Local : Integer;
        begin {Local}
            Local := Level
        end; {Local}
    begin {f}
        if Level > 10 then
            f := f(Level – 1, Local)
        else if Level > 1 then
            f := f(Level – 1, Arg)
        else
            f := Arg { actually call Arg() }

    end; {f}

    function Dummy : Integer;
    begin {Just a placeholder} end;

begin {Main}
    writeln('The answer is:',f(13,Dummy));
end.
```

Figure 9.25 A Pascal Program That Uses Formal Procedures

Furthermore, if a function can be returned as a return value, then frames cannot be allocated on the run-time stack. This is because the frame for a function call may persist even after the invocation returns. Instead, the frames must be allocated on the heap and are subject to garbage collection.

An optimization is to analyze a program so that some frames are allocated on the stack and others on the heap.

Cactus stack Modern computers allow concurrent execution of multiple computations, such as processes, threads, or tasks.

In some cases (such as `fork` in C), a new system-level process is created. Due to the significant operating system overhead, these are called *heavyweight processes*.

A cheaper alternative is to create several threads within a single process. These are called *lightweight processes*. Below is an example of two Java threads:

```
public static void main(String args[]) {
  new AudioThread("Audio").start();
  new VideoThread("Video").start();
}
```

Here are two threads. They share the same run-time stack when they are created. However, later each will push its own new frames onto the stack. Thus, the run-time stack “forks out”. The result is called a *cactus stack*.

A detailed frame layout

Figure 12.9 is the frame layout for the MIPS R3000 compiler. By convention, the first 4 parameters, if they are scalars or pointers, are passed in registers. Additional parameters are passed through the stack.

Registers are partitioned into two groups: *caller-save* and *callee-save* (and no-save). When a subprogram p begins execution, the callee-save registers used by p are saved in the register-save area. When p calls another procedure q, the caller-save registers used by p are saved in the register-save area before jumping to q. The register-save area must be large enough for all calls in p, that is, for all caller-save and callee-save registers.

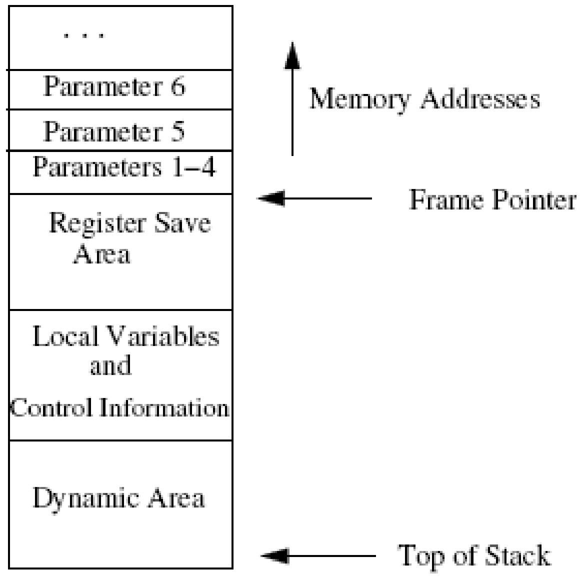


Figure 12.9: Layout for MIPS R3000

JVM did not specify a fixed frame layout for all machine architectures. Different implementations of JVM use different layouts.

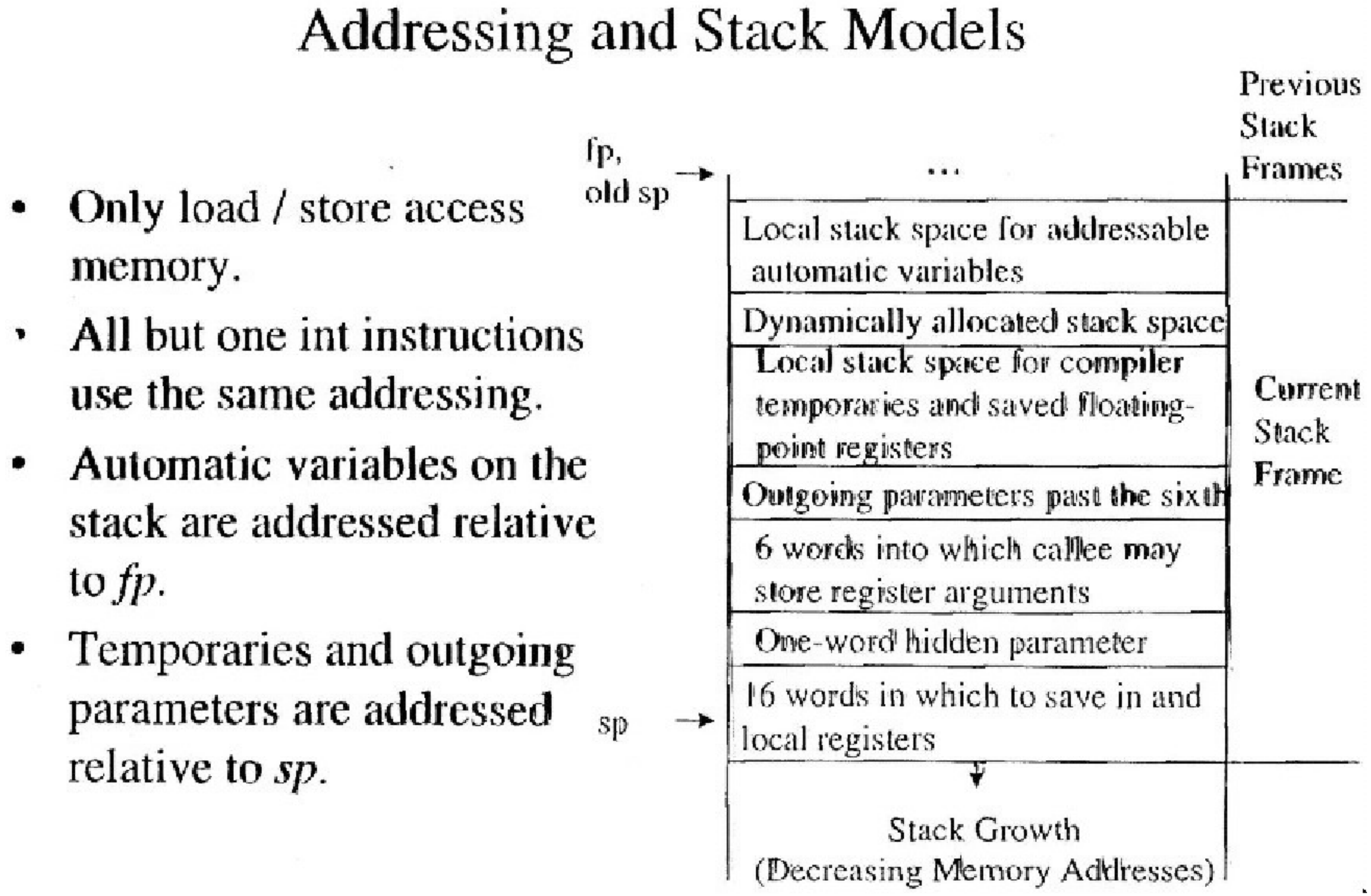
Sometimes, the frame size may grow during execution. For example, sometimes *malloc* could allocate space on the stack, rather than on the heap (because the stack is cheaper to manage) if a detailed program analysis allows such allocation.

Some languages allow *dynamic arrays*.

Some languages allow functions with a variable number of arguments, such as `printf` and `scanf` in C.

For a return value that is a large data structure, we may put the data structure somewhere in the memory and returns a pointer to the space.

A sample AR for one of Sun's compilers

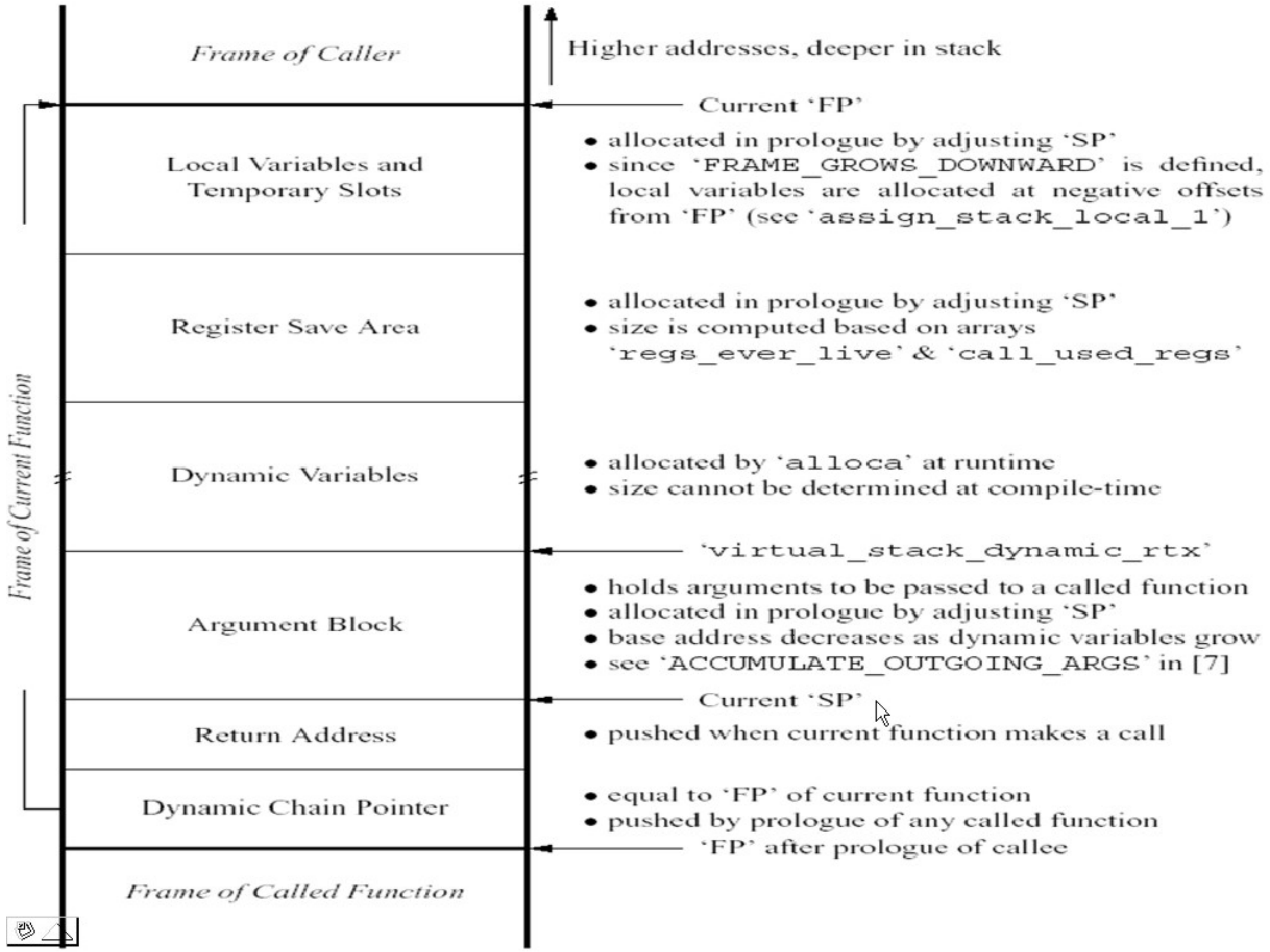


Addressing and Stack Models

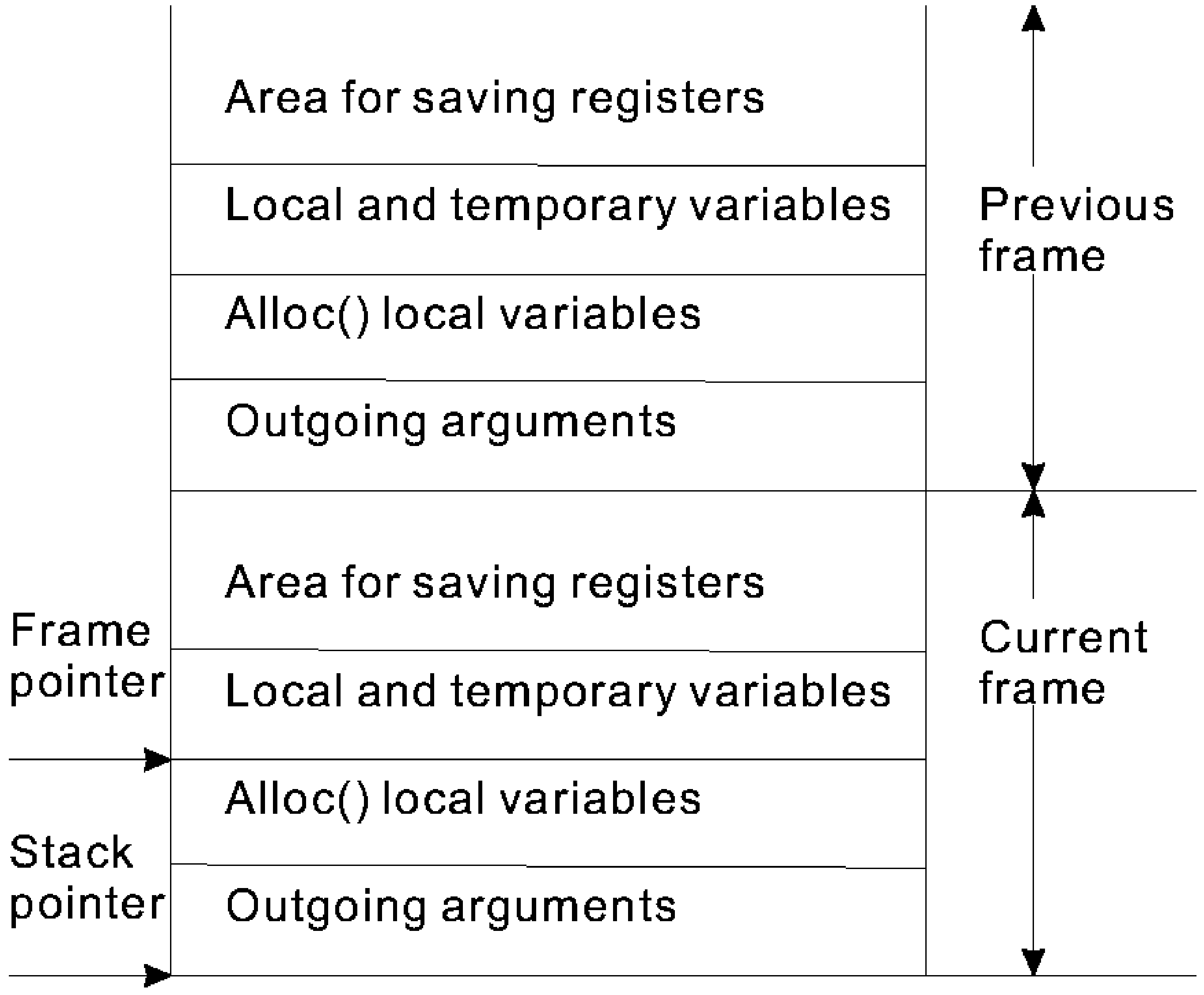
- **Only** load / store access memory.
- **All** but one int instructions use the same addressing.
- Automatic variables on the stack are addressed relative to *fp*.
- Temporaries and outgoing parameters are addressed relative to *sp*.

Previous Stack Frames
fp, old sp
...
Local stack space for addressable automatic variables
Dynamically allocated stack space
Local stack space for compiler temporaries and saved floating-point registers
Outgoing parameters past the sixth
6 words into which callee may store register arguments
One-word hidden parameter
16 words in which to save in and local registers
sp
Current Stack Frame
Stack Growth (Decreasing Memory Addresses)

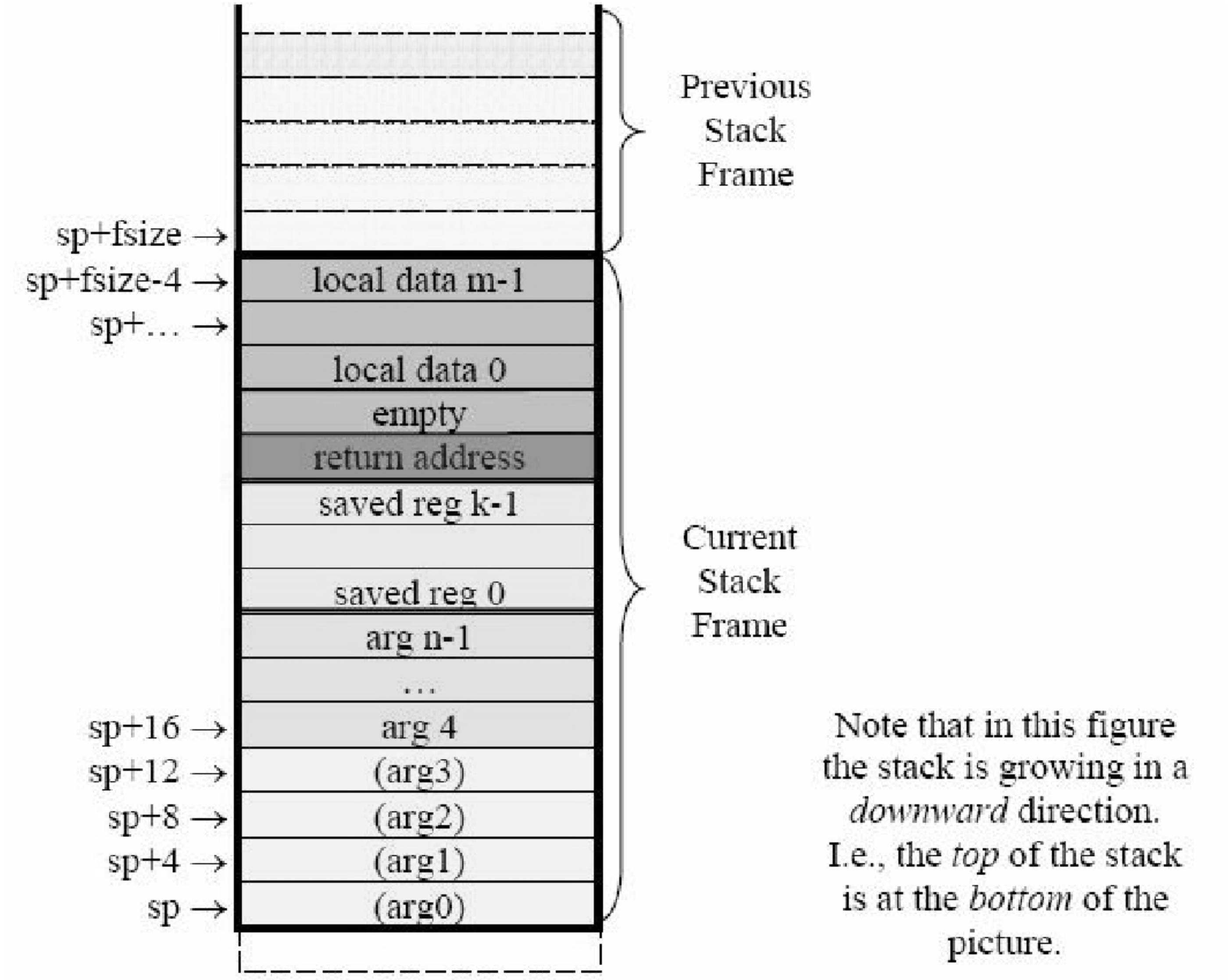
A sample AR for GCC compilers


Frame of Caller
Higher addresses, deeper in stack
Current 'FP'
Local Variables and Temporary Slots
• allocated in prologue by adjusting 'SP'
• since 'FRAME_GROWS_DOWNWARD' is defined, local variables are allocated at negative offsets from 'FP' (see 'assign_stack_local_1')
Register Save Area
• allocated in prologue by adjusting 'SP'
• size is computed based on arrays 'regs_ever_live' & 'call_used_regs'
Frame of Current Function
Dynamic Variables
• allocated by 'alloca' at runtime
• size cannot be determined at compile-time
'virtual_stack_dynamic_rtx'
Argument Block
• holds arguments to be passed to a called function
• allocated in prologue by adjusting 'SP'
• base address decreases as dynamic variables grow
• see 'ACCUMULATE_OUTGOING_ARGS' in [7]
Current 'SP'
Return Address
• pushed when current function makes a call
Dynamic Chain Pointer
• equal to 'FP' of current function
• pushed by prologue of any called function
'FP' after prologue of callee
Frame of Called Function

A sample stack frame for C.



A sample stack frame for MIPS.



Note that in this figure the stack is growing in a *downward* direction. I.e., the *top* of the stack is at the *bottom* of the picture.

§12.3 Arrays

A simple array, such as

```
int a[100];
```

consists of a sequence of identical data objects. The size of an array is

$$size(array) = NumberOfElements * size(Element)$$

Many processors impose an alignment restriction. For example, an integer occupies 4 bytes. Thus, the address of an integer variable must be dividable by 4. Due to the alignment restriction, sometimes padding bytes are added between elements of an array. For example, for the following declaration:

```
struct c { int a; char b; } ar[100];
```

Every “`a`” must be word-aligned. There is a 3-byte padding after every element of the array.

Alignment (and hence padding)

“Not all bytes are equal.”

Data accessed by Intel SSE instructions must be aligned to 16 bytes boundaries. Otherwise, exceptions will occur.

A simplest alignment rule is to align every piece of data and every variable on the 8-byte boundary. This may simplify many alignment analyses. On the other hand, this also wastes a lot of memory.

For alignment, a commonly adopted rule is that a variable is aligned on its size. For example, a `char` variable is byte-aligned; a `short` variable is half-word-aligned; a `int` variable is word-aligned (i.e., 4 bytes).

The alignment of a record is based on the alignment of its largest fields. The size of the whole record is a multiple of the size of its largest (primitive) field. The reason of this alignment is so that we can have an array of records.

Consider the following example, in which the largest field is an `int` (4 bytes).

```
struct {
  char w;
  int x[3];
  char y;
  short z;
}
```

A whole record is word-aligned. `w` is thus, word-aligned. There is a padding of 3 bytes after `w`. `x` is also word-aligned. `y` is also word-aligned. There is a padding of 1 byte after `y`. `z` is half-word-aligned. Therefore, the size of a record is 20 bytes.

Note that if we can re-arrange the order of fields, a record occupies only 16 bytes.

Question 1. Determine the alignments of the following declarations.
(232 bytes, see 483L15)

```
short  a[100];
char   b;
int    c;
double d;
short  e;
struct {
   char f;
   int  g[1];
   char h[2];
} i;
```

Paddings: 10 bytes.

1. 3 bytes after `b`.
2. 2 bytes after `e`.

3. 3 bytes after `f`.
4. 2 bytes after `h[2]`.

`a`: 0-199. `b`: 200. `c`: 204-207. `d`: 208-215. `e`: 216-217. `f`: 220. `g`: 224-227. `h`: 228-229. pad: 230-231

Question 2. Determine the alignments of the following declarations.
(232 bytes, see 483L15)

```
short  a[100];
int    b;
char   c;
double d;
char   e;
struct {
   char f;
   char g[1];
   int  h[2];
} i;
```

Paddings: 8 bytes.

1. 3 bytes after `c`.
2. 3 bytes after `e`.

3. 2 bytes after `g[1]`.

`a`: 0-199. `b`: 200-203. `c`: 204. `d`: 208-215. `e`: 216. `f`: 220. `g`: 221. `h`: 224-231.

Question 3. Determine the alignments of the following declarations.
(132 bytes)

```
short a[37];
char b;
char  c;
double d;
char e;
struct {
   char f;
   char g[1];
   int h[9];
} i;
```

Paddings: 9 bytes.

1. 4 bytes after `c`.
2. 3 bytes after `e`.

3. 2 bytes after `g[1]`.

`a`: 0-73. `b`: 74. `c`: 75. `d`: 80-87. `e`: 88. `f`: 92. `g`: 93. `h`: 96-131.

Case study. In /usr/local/cuda/include/vector_types.h

```
struct __align__(16) double2 {
 double x, y;
}
```

causes `x` and `y` to be 16-byte aligned. Using the `double2` type for the real and imaginary parts of a complex number allows for *coalesced memory access*.

`Storage optimization`.

The layout of an array of characters. Note that there is no padding.

```
char a[16];
```

The layout of an array of (4-byte) integers. Note that there is no padding either.

```
int b[16];
```

Consider an array of structures.

```
struct {
   char a;
   int b;
} e[16];
```

Each structure contains a character field and an integer field. Each structure will take 3-byte padding. However, if the array is stored as two separate arrays, there will be no padding at all. This saves memory space. However, it takes extra time to calculate the addresses of individual elements in the two arrays.

Note that re-ordering the two fields in the structure will not remove the padding.

When copying an array, we may use a series of `load/store` or a simple loop.

Computing the address of `A(I)`

$$\begin{aligned} address\ of\ A(i) &= startaddr + (i - lowerbound) * eltsize \\ &= i * eltsize + (startaddr - lowerbound * eltsize) \end{aligned}$$

What is the difference between the two equations?

The subexpression $(startaddr - lowerbound * eltsize)$ is called the *virtual origin* of the array.

We may need to generate code to test whether the index is within the proper range.

A better compiler can selectively generate the range test by doing a comprehensive flow analysis on the value of the index `I`.

Here we make an assumption: array elements are placed consecutively. Advanced compilers may distribute array elements in different ways in order to improve performance.

An optimization in calculating the element addresses in an array.

Arrays are frequently associated with loops, which are an important target of optimization. A typical loop has the following form:

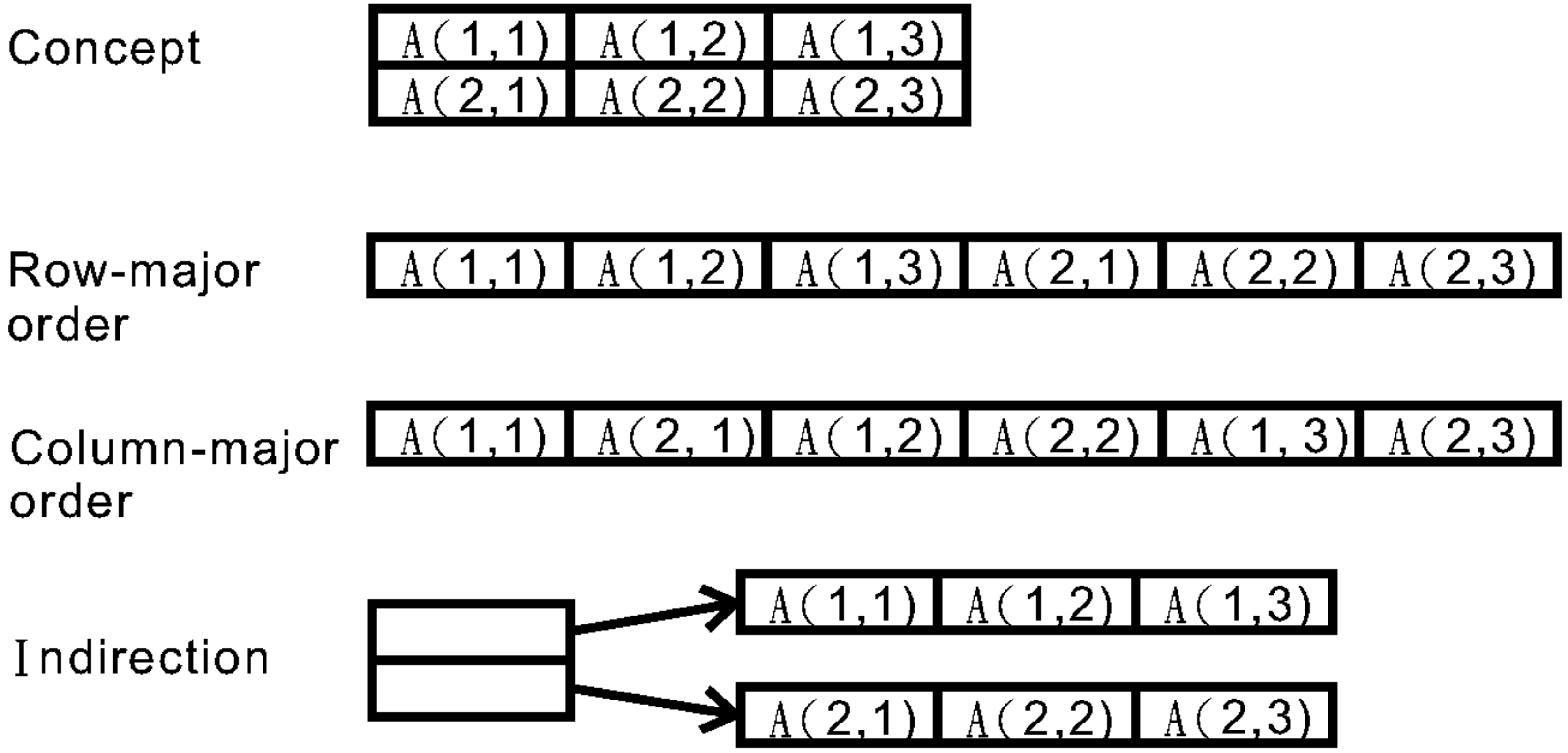
```
for i = 1 to 100 do
    . . . A[i] . . .
end
```

Note that the elements are accessed consecutively, that is, `A[1]`, `A[2]`, `A[3]`, ..., `A[100]`. Though we may calculate the addresses of the elements separately, it would be advantageous to add the offsets to a previous address. That is, the address of `A[2]` is obtained by adding 4 to the address of `A[1]`, etc. The code generated for the loop would be

```
p := address of A[0]
for i = 1 to 100 do
    p := p + 4
    use p as the address of A[i] inside the loop
    . . . A[i] . . .
```

```
end
```

Common array layouts



Question.

When are different layouts (for arrays) useful? Tiled. See GEMM.

Array bounds checking

Java always checks the index of an array element is within the allowed bounds. We need to check both the lower bound and the upper bound. Redundant bound checks may be optimized away.

One trouble is that an array is often treated as a pointer to its first element. There is no way to know the upper bound of a pointer.

Sometimes, in order to ease the computation of element addresses, many C and C++ programs intentionally violate the bound checking by initializing the pointer one element before the array.

When an array is passed as a parameter, we need to know not only the address but also the size and bounds. This information together forms an array descriptor, which is also called a *dope vector*.

In C, C++ and Java, the lower bound of an array index is always 0. This may lead to clumsy code.

§12.3.2 Multi-dimensional arrays

Multi-dimensional arrays are treated as arrays of arrays. For example,

```
int matrix[][] = new int[5][10];
```

The allocation of such arrays of arrays are shown in Figure 12.11.

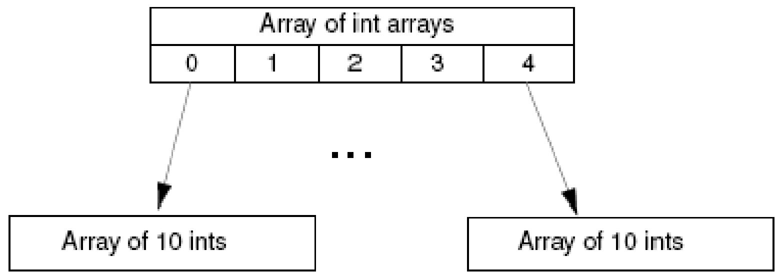


Figure 12.11: A Multidimensional Array in Java

Other languages allocates a block of memory for the whole arrays. The elements could be in row-major or column-major order.

For 2-dimensional or higher-dimensional arrays, we need to be careful of row-major or column-major (FORTRAN) order. This affects the calculation of an element's address.

The array `A(2, 3)` may be stored as

1. `A(1, 1), A(1, 2), A(2, 1), A(2, 2), A(3, 1), A(3, 2)`
2. `A(1, 1), A(2, 1), A(3, 1), A(1, 2), A(2, 2), A(3, 2)`

See Figure 12.12.

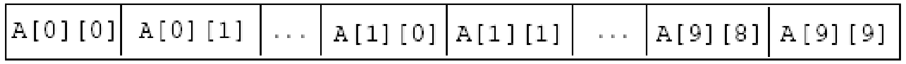


Figure 12.12: Array `A[10][10]` Allocated in Row-Major Order

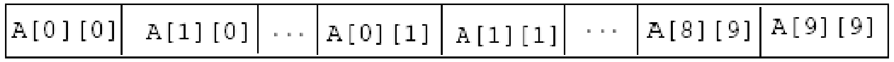


Figure 12.13: Array `A[10][10]` Allocated in Column-Major Order

Assume an array is declared as follows:

```
SomeType A[0..n][0..m];
```

Assume the array is stored in the row-major order. The address of the element `A[i][j]` is

$$addr(A[i][j]) = addr(A) + i * m * eltsize + j * eltsize$$

Matrix transposition is a common operation in array manipulation. Note that the address of `A[i][j]` in row-major order is exactly the address of `AT[j][i]` in column-major order. See Figure 12.14.

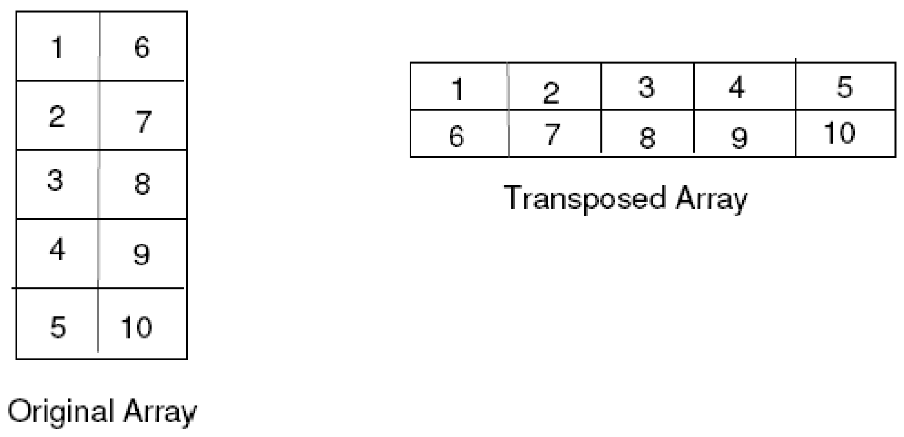


Figure 12.14: An Example of Array Transposition

By changing the computation of element addresses, sometimes it is not necessary to explicitly transpose a matrix.

§12.4 Heap management

Sometimes, the size of a data structure can be decided only at run time. In this case, we use dynamic allocation. `new` and `malloc` allocate storage on the heap during program execution. Here are some examples.

```
malloc(50)                (in C)
new(P)                    (in Pascal)
new Aclass                (in Java)
allocate                  (in PL/1)
```

The life time of a dynamically allocated data object is usually un-predictable. Hence it cannot be allocated statically nor on the stack. It is allocated on a heap.

Heap management, including allocation, deallocation, and garbage collection, is expensive. Heap is more flexible and more complicated (than stack and static storage) and hence more expensive to manage. Objects should not be allocated on the heap if they can be allocated elsewhere. For example, dynamic arrays are dynamically allocated but they can be allocated on the stack due to their life span follows a simple rule.

§12.4.1 Allocation mechanism

1. explicit allocation
   `allocate` in PL/1
   `new(p)` in Pascal and Ada
2. implicit allocation
   `Str1 + Str2` in Java
   `cons` in LISP
   `"abc" "def"` in SNOBOL
3. no allocation (in the language itself)
   `malloc(20)` in C (All is done in the library. The compiler simply generates code to call the library function.)

Allocated space is almost always single-word or double-word aligned. A heap block starts with a 1-word header. A heap block should not be too small to avoid fragmenting the heap.

After a series of allocations and deallocations, a heap manager may *compact* the heap by moving all the live blocks together. This results in a single large free block.

All the free blocks may be kept on a *free-space list*. Blocks of space are allocated and freed gradually.

When two free blocks are adjacent, they should be merged into a single block. We may add *boundary tags* to each block during allocation so that it would be easier to identify adjacent free blocks.

Instead of boundary tags, we may also keep the free list according to the address order.

The trouble with dynamic allocation lies in storage reclamation (that is, *garbage collection*). Either the programmer is responsible for collecting the garbage (as in C) or the system automatically collects garbage (as in Java).

1. No deallocation
   good if most heap objects stay useful.
   good if there is a really big virtual memory.
   A user could maintain a private free-list.

2. explicit deallocation
   `free( )` in C
   `dispose( )` in Pascal
   Users free space explicitly; heap manager keeps track of free space.
   There are two problems:

   a. dangling pointer problem : free some memory that should not be freed.

   b. memory leak : Did not free some memory that should be freed.

Should the implementation check dangling pointers?

3. implicit deallocation (garbage collection)
   Useless space is automatically recovered.
   E.g., Java and Lisp.

§12.4.3 Automatic garbage collection

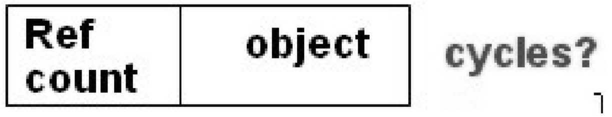
Garbage collection is a tedious and error-prone task. The most serious problem caused by erroneous garbage collection is the *dangling pointer problem*, in which a pointer holds an address of a memory block that is already freed.

```
var p, q : pointer to real;
. . .
new(p);
q := p;
dispose(p);
^q := . . .
```

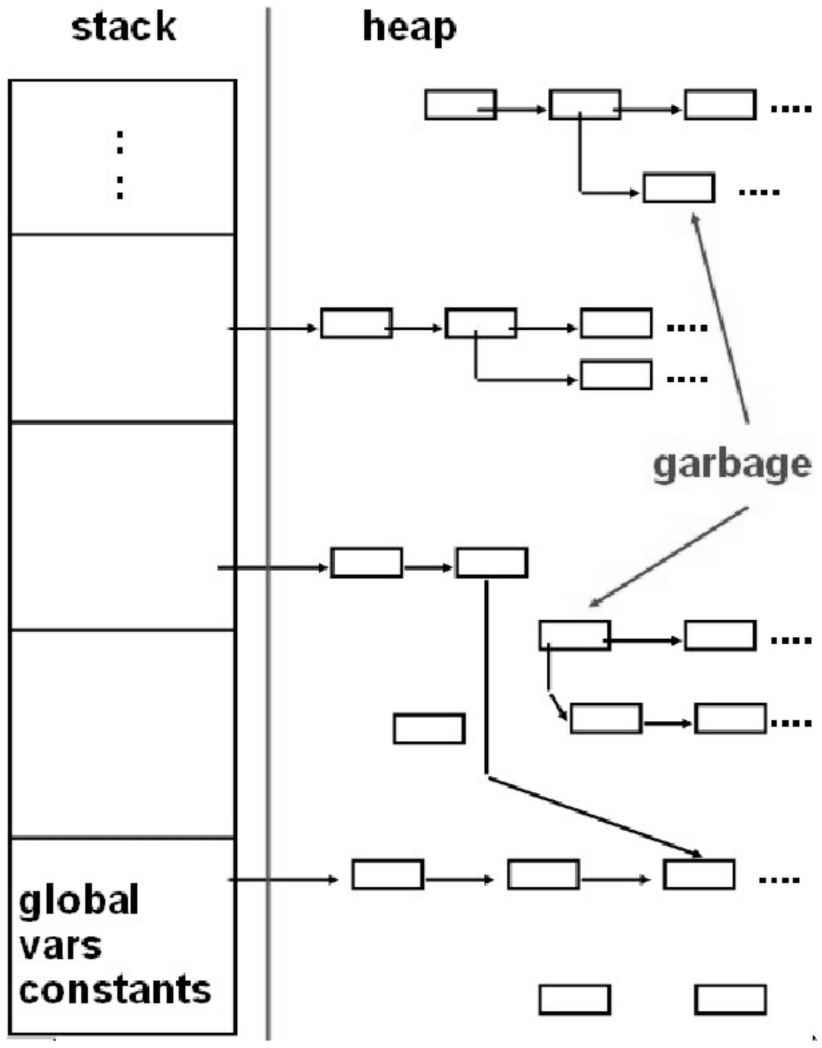
Three approaches to garbage collection:

single reference At most one pointer to a heap object, e.g., strings.
Not good for complex linked structures.

reference count For each object, keep a count of the number of pointers to the object. Free the object when its count is 0.



mark-sweep-compact Find all global pointers (including those in stack).
Mark all heap objects reachable from the global pointers.
Free all unmarked heap objects.
Compact remaining objects.


stack
heap
⋮
garbage
global
vars
constants

Reference counting has trouble with circular heap structures.

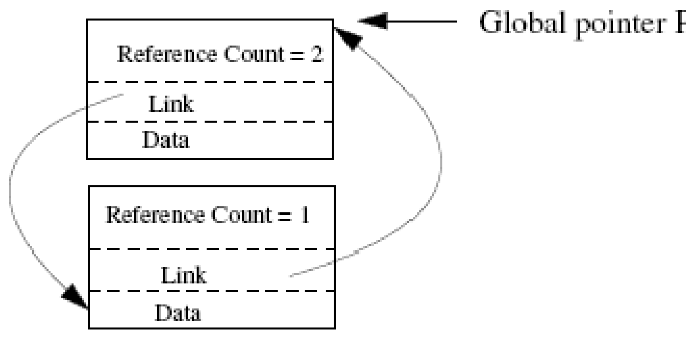


Figure 12.15: An Example of Circular Structures

The good thing about reference counting is that it is *incremental.* This is good for real-time applications. However, the total overhead of reference counting is higher than a mark-sweep collector because it is too *eager* to modify the counters. A *lazier* approach might reduce the total overhead.

We may also set up a maximum for counters, say 7, 15, or 31. When a counter reaches such a maximum, it is consider never-dying.

Compiler must generate enough information concerning where global pointers are.

Run-time routines perform marking, sweeping, and compaction.

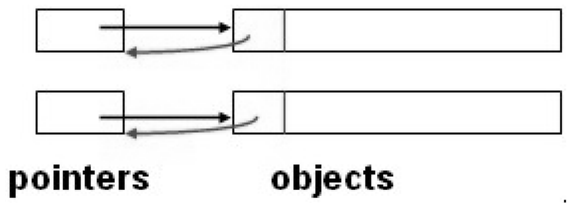
Garbage collection is activated only when heap space is exhausted, not whenever references are created or destroyed.

Easy for uniform languages, e.g. LISP. Difficult for complex languages,e.g. Ada.

Compaction is used to avoid memory fragmentation.

garbage collection for single-reference objects (e.g. Ada/CS strings)

hand-shaking convention



Sweep through heap objects. Keep the heap object if

1. It points to an AR in use.
2. The corresponding pointer in AR points back to the object.

There is slight chance for uncollected garbage. Uninitialized pointers may be mistaken as a valid pointer.

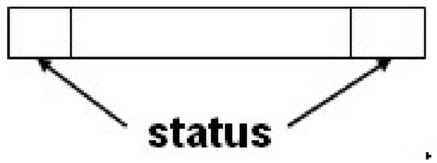
Solution.

1. Initialize all pointers when they are created.
2. Check validity when referencing.

Managing heap space: similar to hard disk space management in OS.

1. first-fit
2. best-fit
3. circular-first-fit

We may add *boundary tag* in each block for combining adjacent free chunks.



Usually, there are only a few different sizes of allocated chunks. We may use different heaps for differently-sized chunks and manage each heap with a bit map.

Garbage collection is usually done in three phases: mark, sweep and compact.

In the mark phase, all the accessible memory blocks are marked. Then all memory blocks that are not marked are swept together. Finally, the marked memory blocks are moved around so that they stay together.

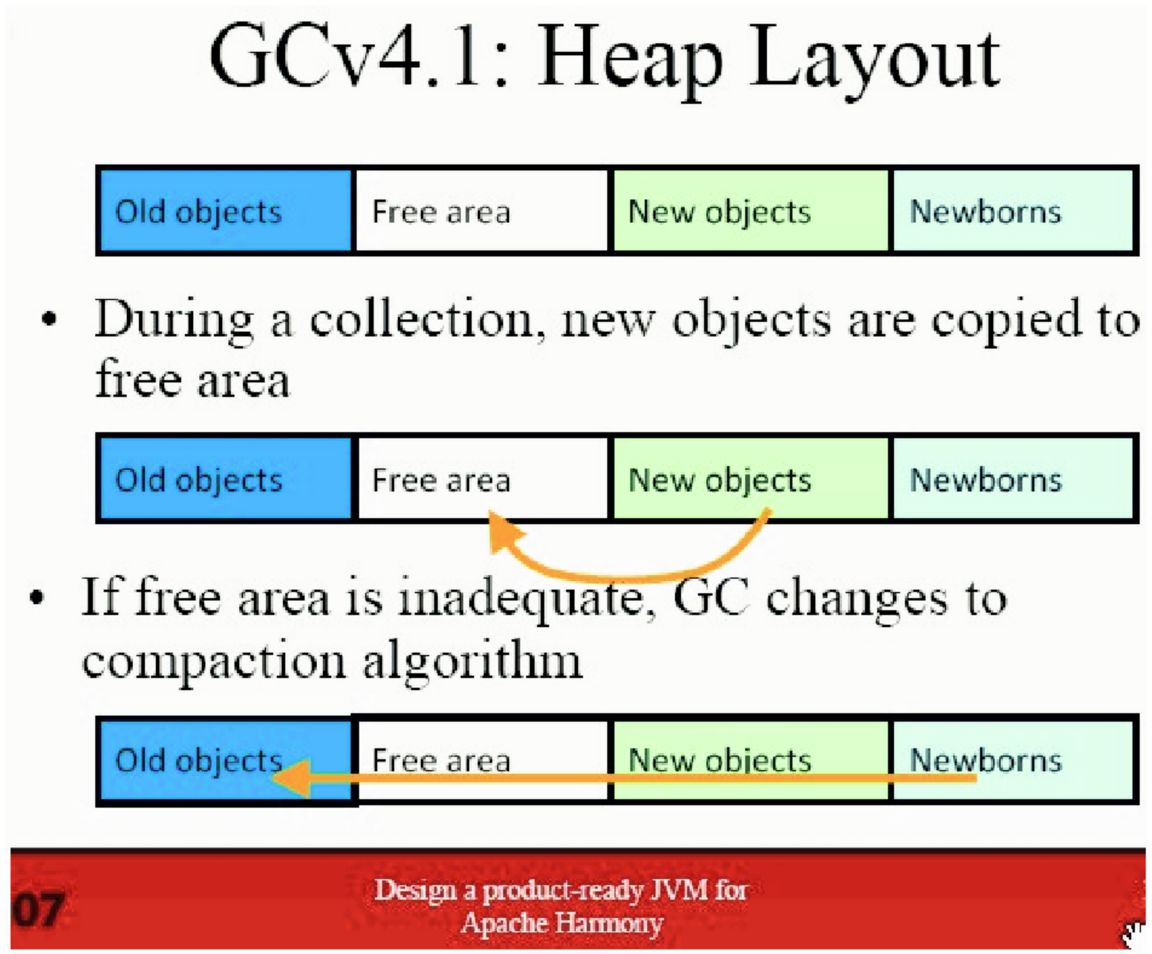
The compiler needs to generate enough information to facilitate garbage collection during run time.

Copying collectors make use of two semi-spaces.

Generational collectors make use several generations.

Case study. Harmony (ossummit07-drlvm_overview.pdf) comes with 3 garbage collectors:

- GC v4.1:
  - copying collector with compaction fallback
  - simple, sequential, non-generational
- GC v5:
  - copying collector with compaction fallback
  - parallel, generational
- Tick:
  - on-the-fly mark-sweep-compact
  - concurrent, parallel, generational


GCv4.1: Heap Layout

Old objects
Free area
New objects
Newborns

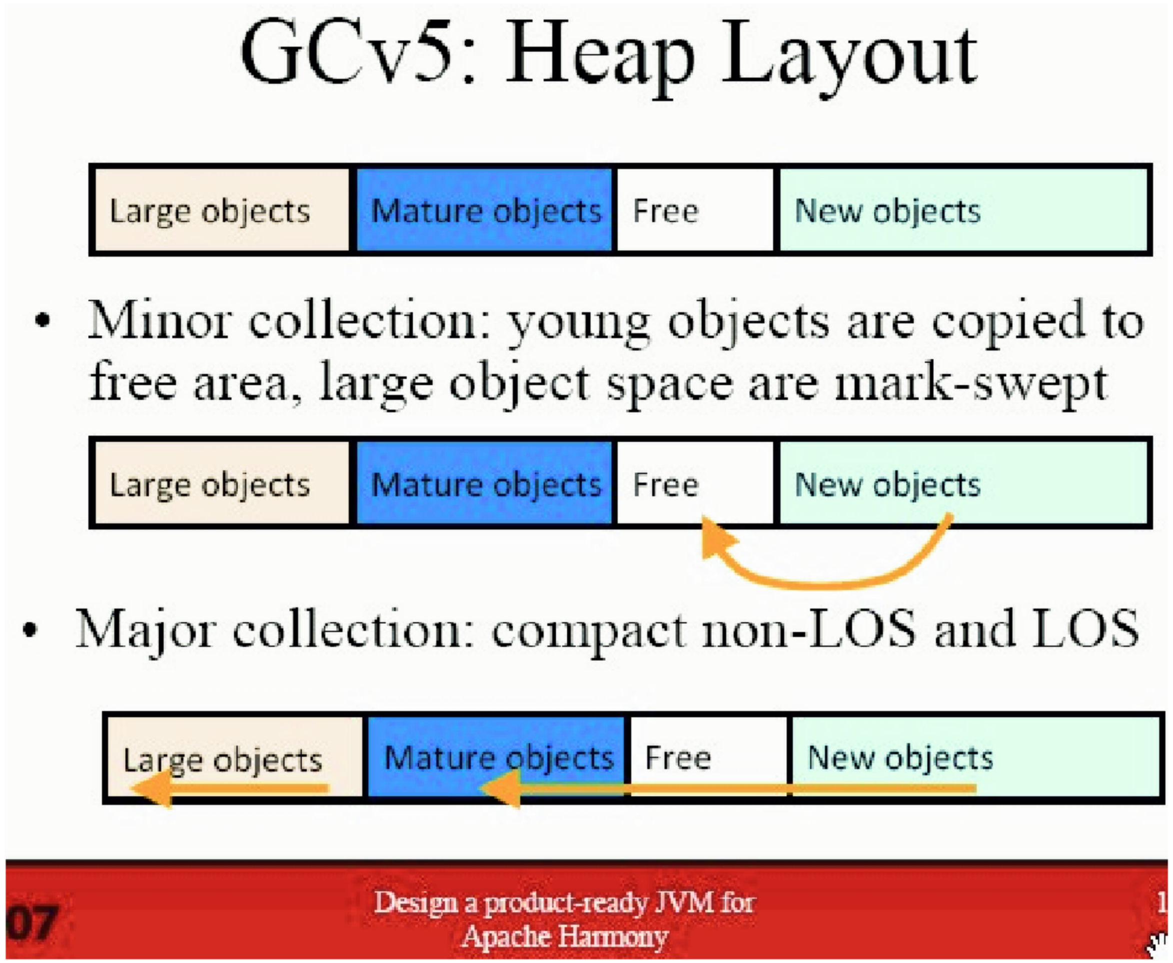
- During a collection, new objects are copied to free area

Old objects
Free area
New objects
Newborns

- If free area is inadequate, GC changes to compaction algorithm

Old objects
Free area
New objects
Newborns
07
Design a product-ready JVM for Apache Harmony

Figure 8: Garbage Collection, v4.1, in Harmony



GCv5: Heap Layout

Large objects
Mature objects
Free
New objects

- Minor collection: young objects are copied to free area, large object space are mark-swept

Large objects
Mature objects
Free
New objects

- Major collection: compact non-LOS and LOS

Large objects
Mature objects
Free
New objects
07
Design a product-ready JVM for Apache Harmony
1

Figure 9: Garbage Collection, v5, in Harmony

§12.5 Region-based memory management

A *region* is like a heap, an area of space in which new objects are allocated. A region may grow as new objects are allocated but never shrinks. A region is deallocated all at once; individual objects in a region are never freed separately.

A program needs to add region-creation and region-deletion operations explicitly. When allocating a new object, `new` should specify the region for the allocation.

For example, assume 3 procedures A, B, and C. A calls B and B calls C. Each procedure begins by creating a new region and ends by destroying that region. The regions can be allocated on top of the stack.

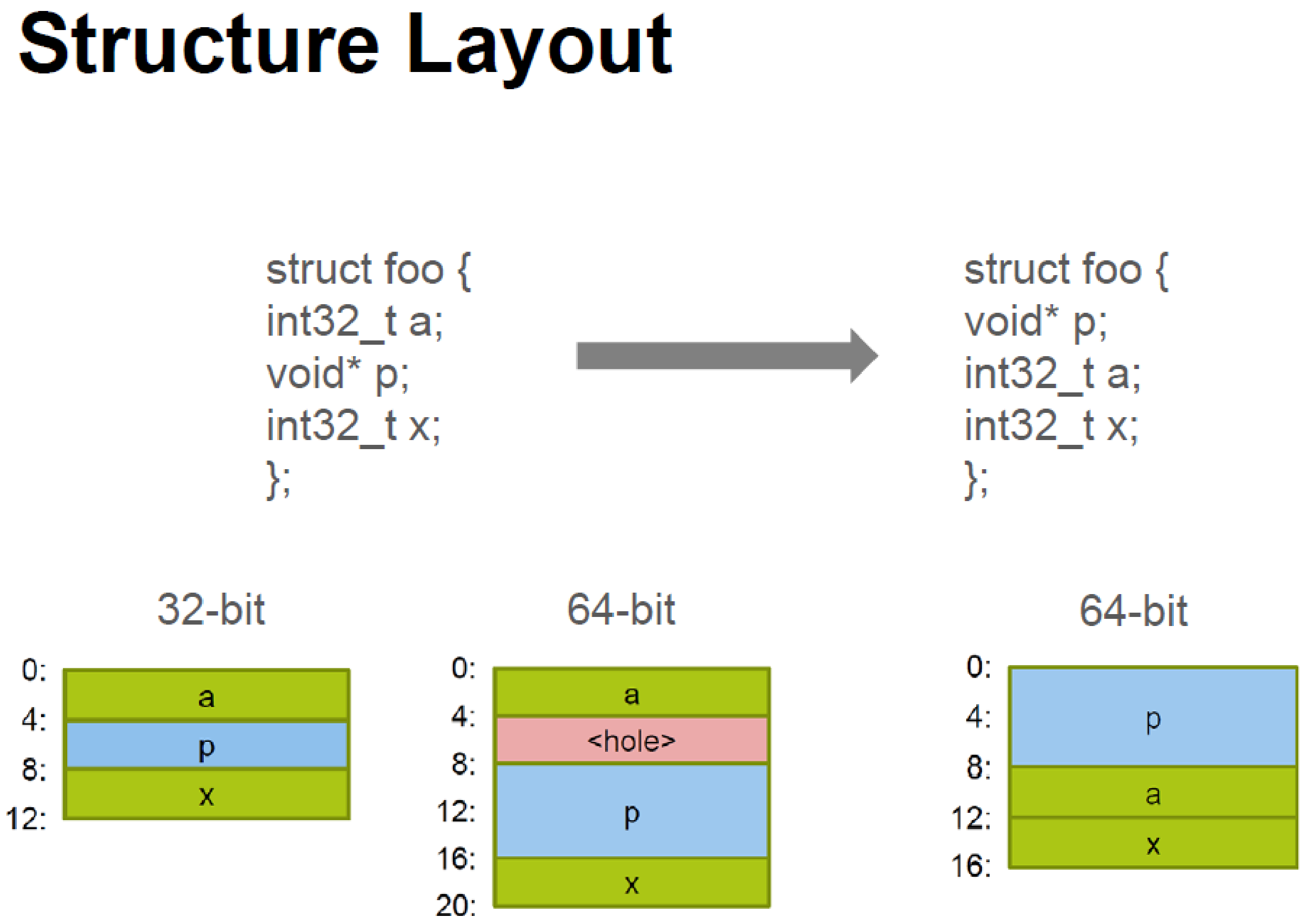
Alternatively, C can allocate space in A's region.

Regions can be organized as a stack and have lexically scoped lifetimes so that deallocation is quick and easy.

In order to use regions, a compiler needs to analyze the lifetimes of objects (called *region analysis*) and determine the placement of regions and objects.

Languages may be designed so that region analysis can be done easily. E.g., some real-time Java systems, many ML compilers and the Cyclone language.

Appendix. Structure layout.



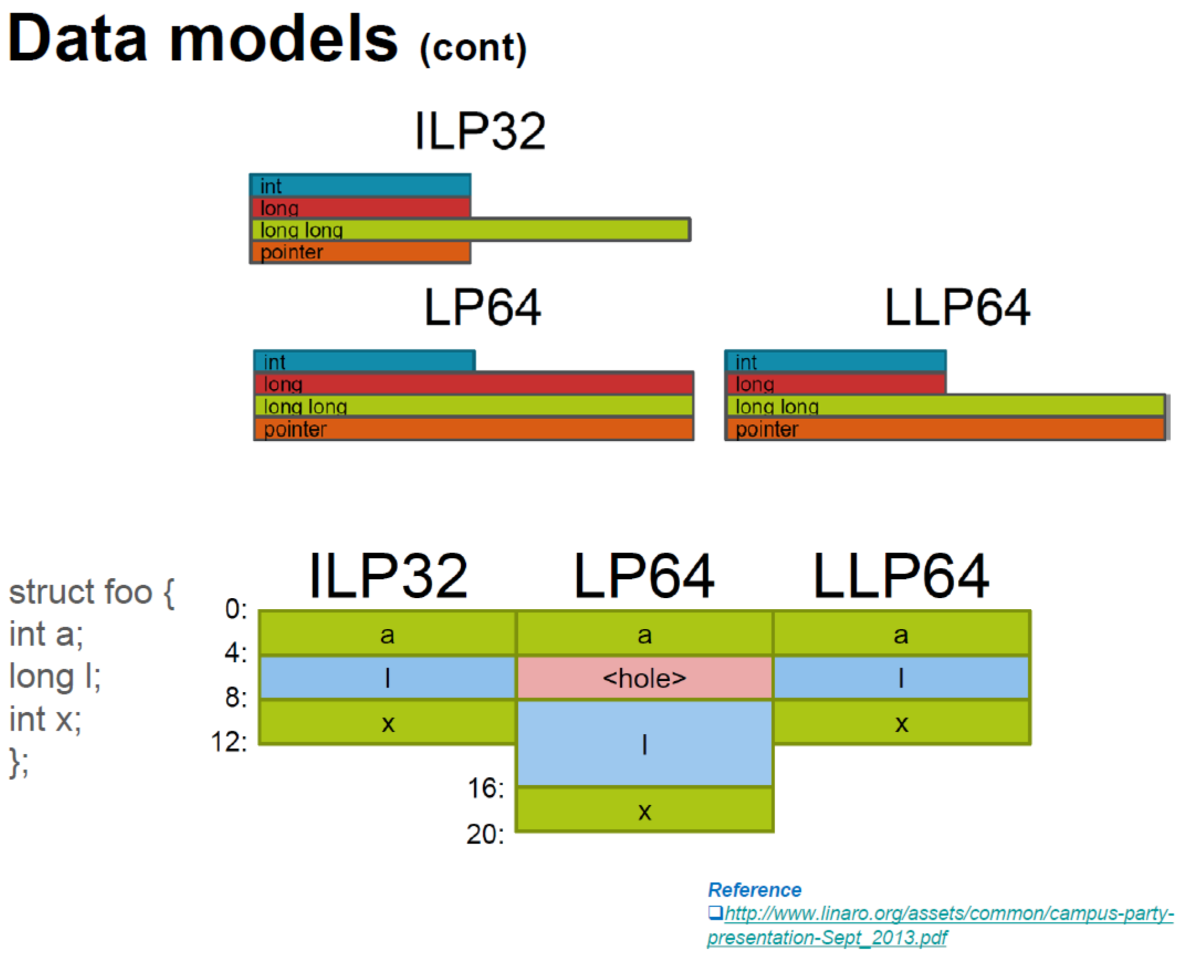
Structure Layout

```
struct foo {
int32_t a;
void* p;
int32_t x;
};
```

```
struct foo {
void* p;
int32_t a;
int32_t x;
};
```

Figure 10: Structure layout.

Data model and layout in AArch64.



```
struct foo {
int a;
long l;
int x;
};
```

Reference
❑http://www.linaro.org/assets/common/campus-party-presentation-Sept_2013.pdf

Figure 11: Data model and layout in AArch64.

This is the end of Chapter 12.

Figure 12: This is the end of Chapter 12.