Compiler Technology of Programming Languages

Code Generation

Part I:Run-Time Support

Prof. Farn Wang

Storage Allocation

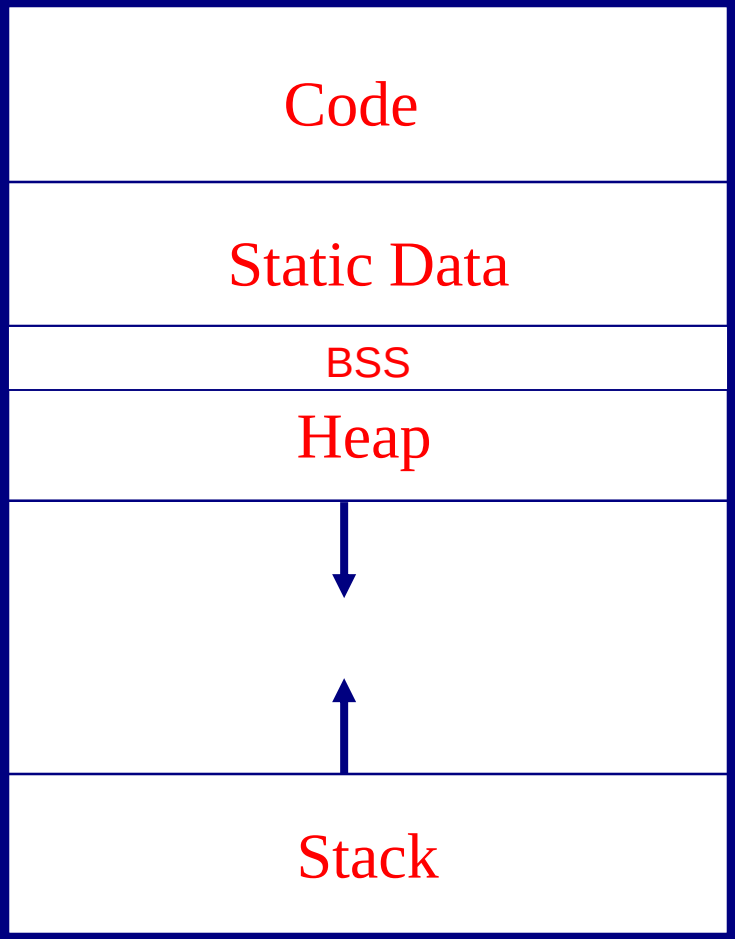
- Static allocation
  - Storage allocation was fixed during the entire execution of a program
- Stack allocation
  - Space is pushed and popped on a run-time stack during program execution such as procedure calls and returns.
- Heap allocation
  - Allow space to be allocated and freed at any time.

Space efficiency
Recursive procedures

	Static	Stack	Dynamic
Allocation time	Compile time	Procedure invocation	malloc() at runtime
Variable lifetime	Program start execution to end	Procedure call to return	From Alloc() to Free()
Space efficiency	Less efficient	Space reuse is high	Depends on user management
Implementation	Data segments	Runtime stack	Heap
Access efficiency	Fast (gp+offset)	Fast (fp+offset)	Indirect reference Less locality
Compiler optimization	Easier	Easier	More difficult

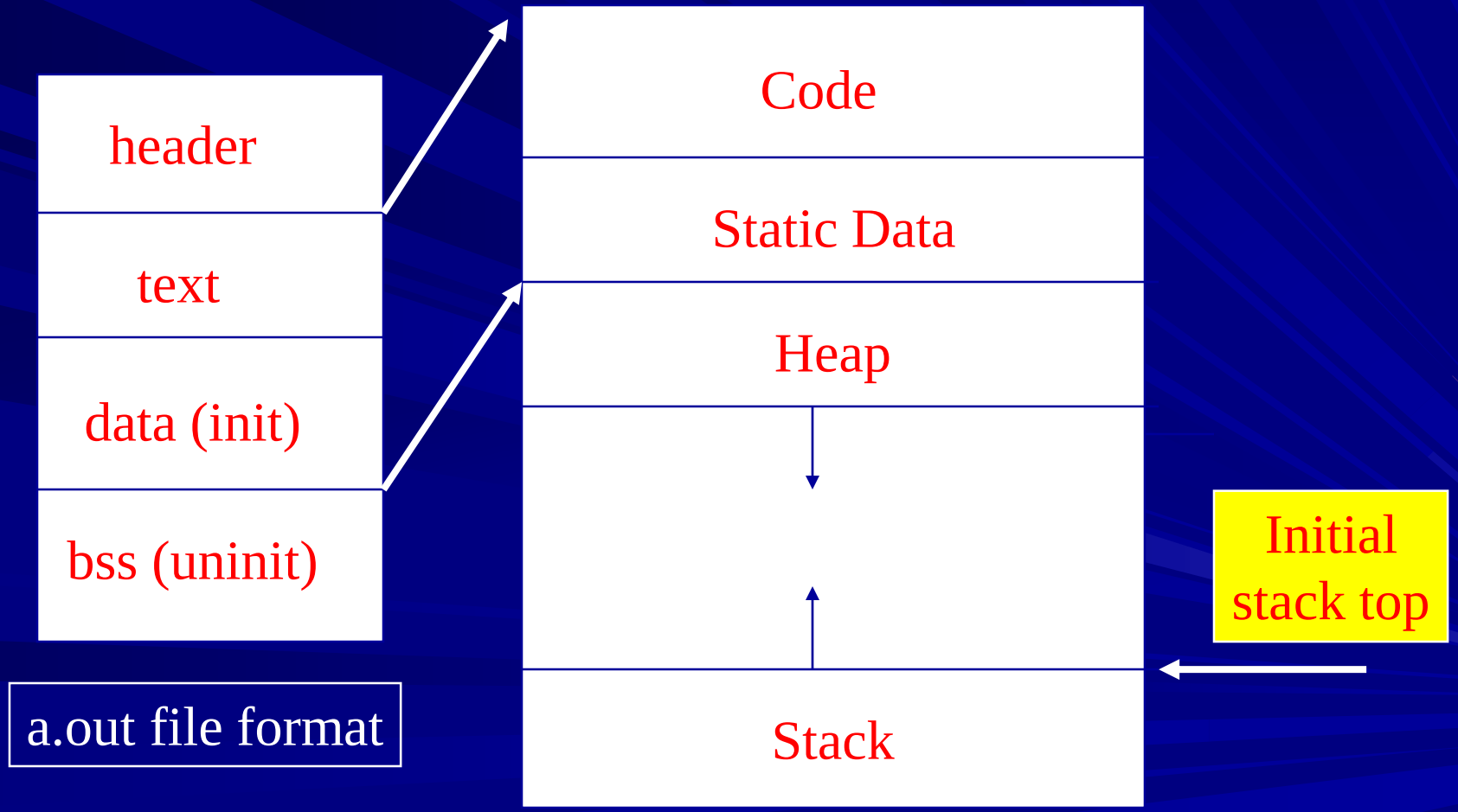
Storage Organization

- Code
- Static data
- Stack
- Heap



Virtual address

Storage Organization



ELF Object File Format

ELF:
Executable and Linkable Format

A common standard file format on Unix or Unix-like sys for executables, object code, shared lib, and core dumps

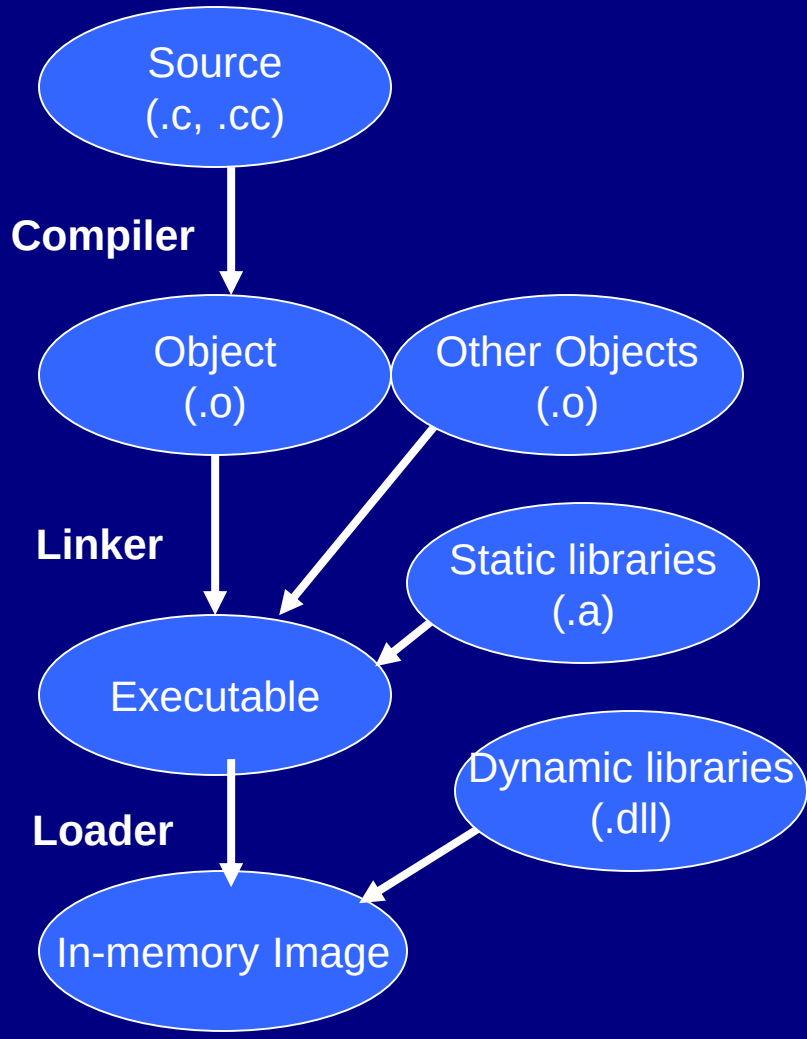
%readelf –a a.out

Example Header (64byte)

Magic number 4 bytes: 0x7F E L F (7F 45 4c 46)
1 – 32 bit, 2 – 64 bit
1 – little endian, 2 – big endian
ABI: System V, HP-UX, NetBSD, Linux, Solaris, AIX, FreeBSD,…
1 – relocatable, 2 – executable, 3 – shared, 4 -- core
ISA: SPARC, x86, MIPS, PPC, ARM, IA-64, x64, AArch64, RISC-V,…
Address of the entry point
Points to the section header table
Size of the header
Number of entries
Size of the section header table
…..

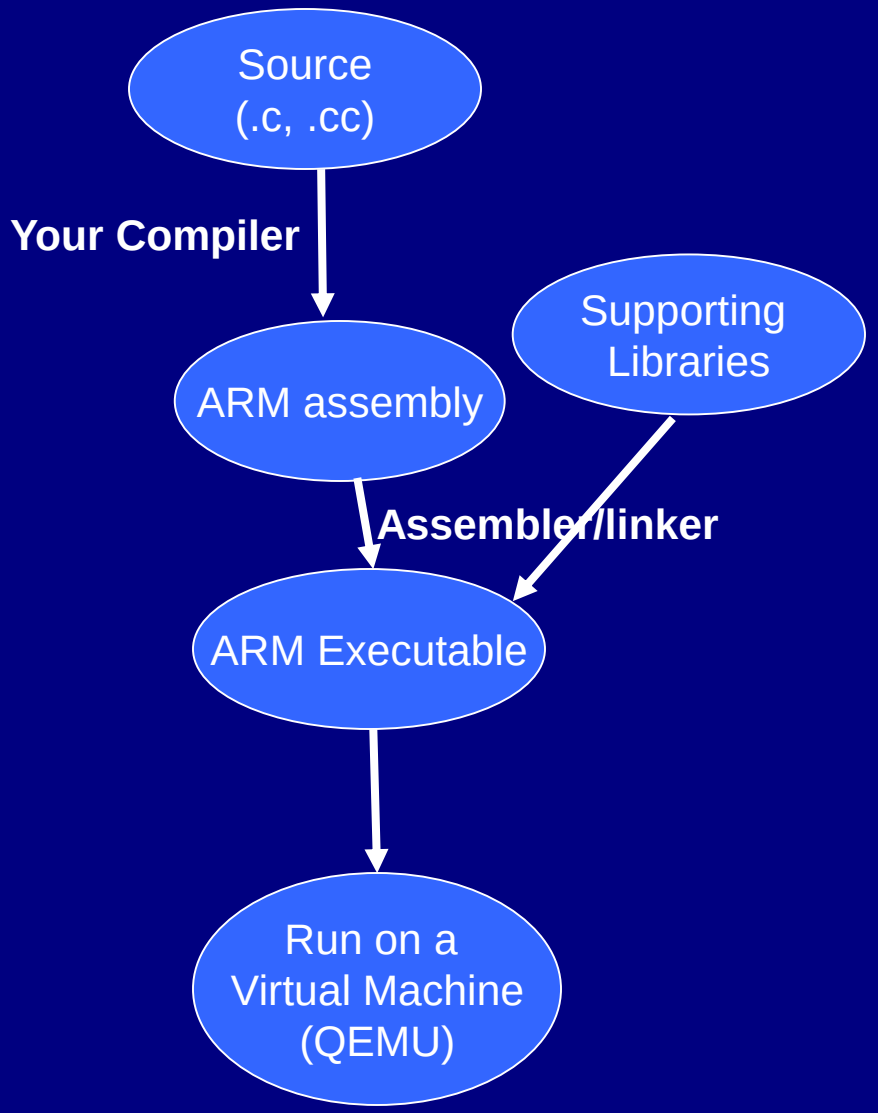
Compile-Link-Load

- *Binding* is the process of mapping *names* to *addresses*
- Most compilers produce *relocatable object code*
  - Addresses relative to *zero*
- The linker combines multiple object files and library modules into a single executable file
  - Addresses also relative to *zero*
- The Loader reads the executable file
  - Allocates memory
  - Resolves names of dynamic library items



Project III

- All source functions are in one file, your compiler will translate the source file into one assembly code file.
- Assembly code file and supporting libraries (for I/O) will be assembled and linked together by an assembler. The output is an ARM executable.
- This ARM executable will be run on a Virtual Machine (e.g. QEMU).



Static Allocation

- Bindings between names and storage are fixed at compile-time.
- The values of local names are retained across activations of a procedure.
- Addressing is efficient
- Limitations:
  - No recursive procedures
  - No dynamic data structures

*In old days, when only static allocation is allowed, programmers often use **OVERLAY** to overcome space limitation.*

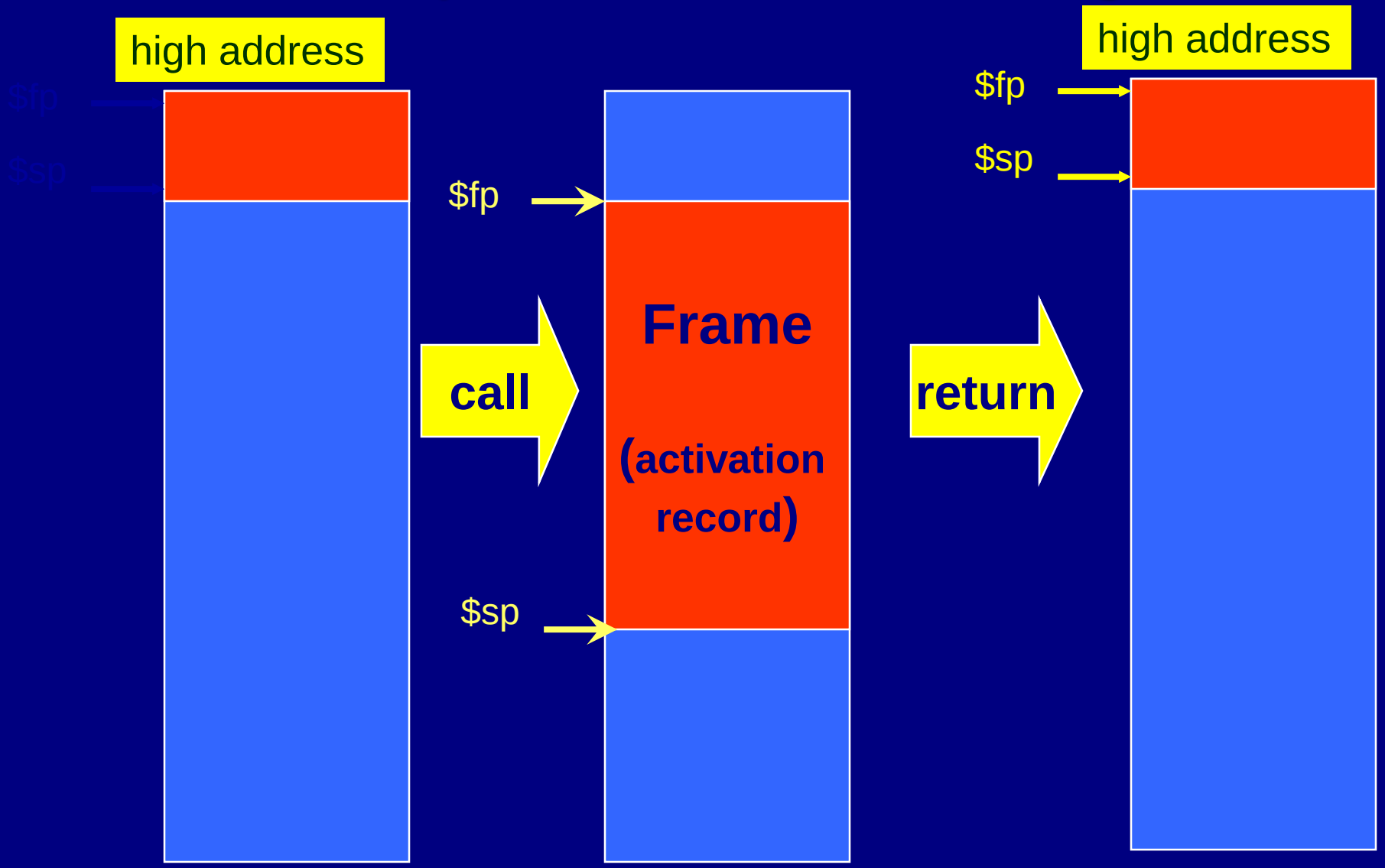
Activation Record (Frame)

in calling stack

- Temporary values
- Local variables
- Saved machine states (arch states)
- Parameters
- Return value
- Access link (static link)
- Control link (dynamic link)

The sizes of almost all fields can be determined at compile time – except for dynamic arrays or other variable length data objects

Frame Space on the CallingStack



Stack Allocation

- Recursive procedures require stack allocation.
- Activation records (AR) are pushed and popped as activations begin and end.
- The offset of each local data relative to the beginning of AR is stored in the symbol table.

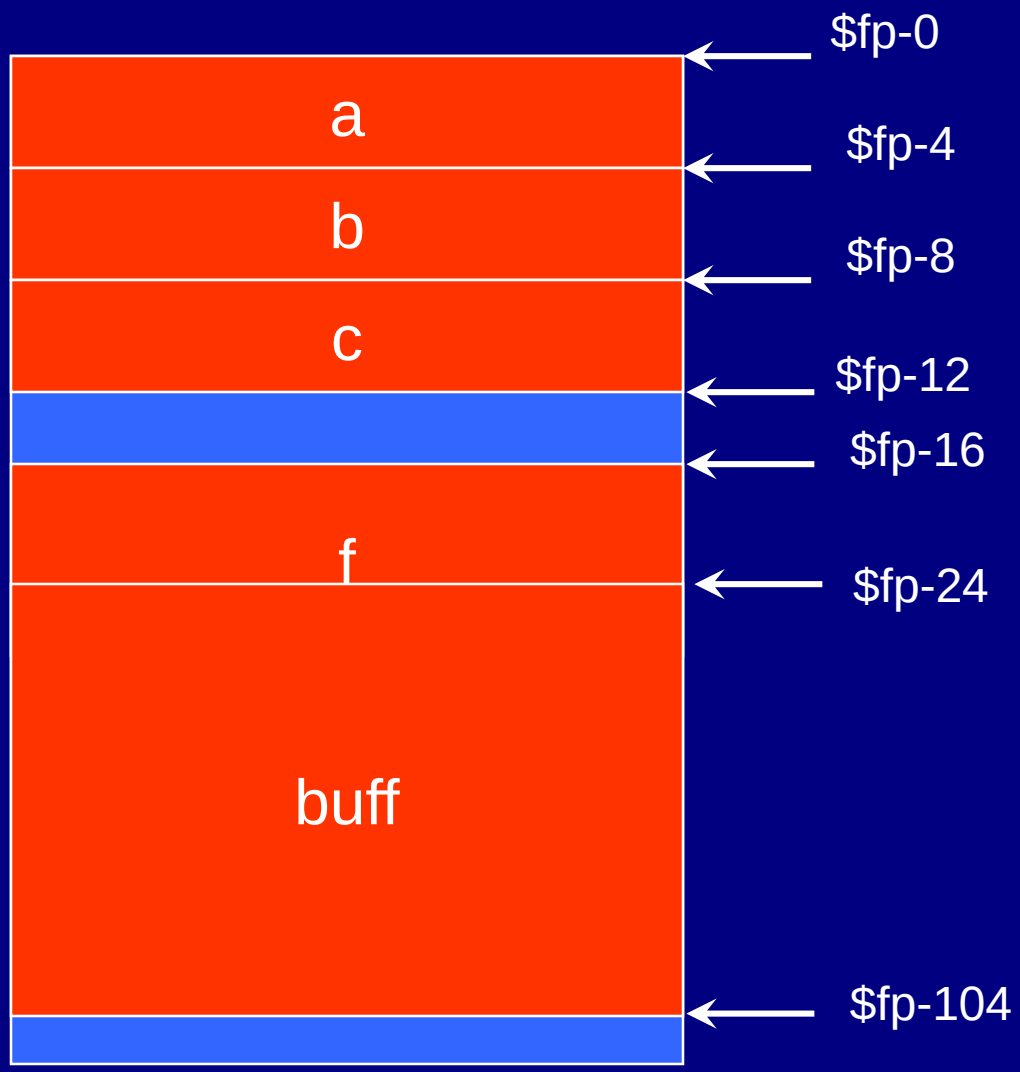
```
float f(int k) {
  float c[10],b;
  b = c[k]*3.14;
  return b;
}
```

Return value	offset = ?
Parameter k	offset = ?
Local c[0]	offset = 40
Local b	offset = 44

3.14 goes to a literal pool, not to AR. Why not?

Local variable allocation and layout

```
Int a,b,c;
Double f;
Char buffer[80];
```



Static Allocation and Stack Allocation

- In project II, we do not separate global variables from local variables.
- Now we need to allocate storage for them differently:

  static allocation for globals,

  stack allocation for locals.

First of all, how do we distinguish globals from locals?

One possibility: if (scope == 0)
global (i.e. static allocation)

Compile-Time Layout of Local Data

- The amount of storage needed for a name can be determined from its type.
- Basic data types: char, real, integer, .. etc
- Aggregates: record (struct), array, .. etc.
- Address for local data objects: fixed length data is laid out as declarations are processed.
  - The compiler tracks the number of memory locations allocated as the relative address (or offset) for local data objects.

Alignment requirements

- an array of 10 chars may end up allocating 12 bytes.
- For cache and page efficiency!
- For portability of object code!
  - Some CPUs require this.

Example:

```
Struct S1 {
        int i;
        char c;
        int k;
}
```

12 bytes are allocated for this struct

Quiz

- If a C struct occupies 9 bytes, what would the nature boundary this struct should be aligned at?
  - 1) 10
  - 2) 12
  - 3) 16
  - 4) 64

1, 2, 3 are all possible, depending on the member's types

Alignment Example

```
Struct S1 {
        long long i;
        char c;
        short int k;
}
```

How many bytes are allocated for this struct?

Alignment Example

```
Struct S1 {
        long long i;
        char c;
        short int k;
}
```

16 bytes are allocated for this struct

Struct S1 ArrayB[100], if 12 bytes are allocated for each struct, ArrarB[1].i would be misaligned.

Stack Alignment

How to ensure the stack top (or fp, frame pointer) is properly aligned?

- ABI (Application Binary Interface) for ARM requires that the stack must be 8 byte aligned on all external interfaces, such as calls between functions in different source files.
- Other ABI may require the stack top to be 32 or 64 byte aligned.

Static data layout in our project

```
int a, b[100];

int main ()
{
….
}

int c;

…
```

```
.data
a: .word    // int a;
b: .space 400 // int b[100];
m: .ascii “enter a number”
```

```
.text
main:
```

```
.data
```

Static *vs.* Stack Allocation

If (global)

if (int scalar) print (var name, “:”, “.word”);

if (float scalar) print (var name, “:”, “.float”);

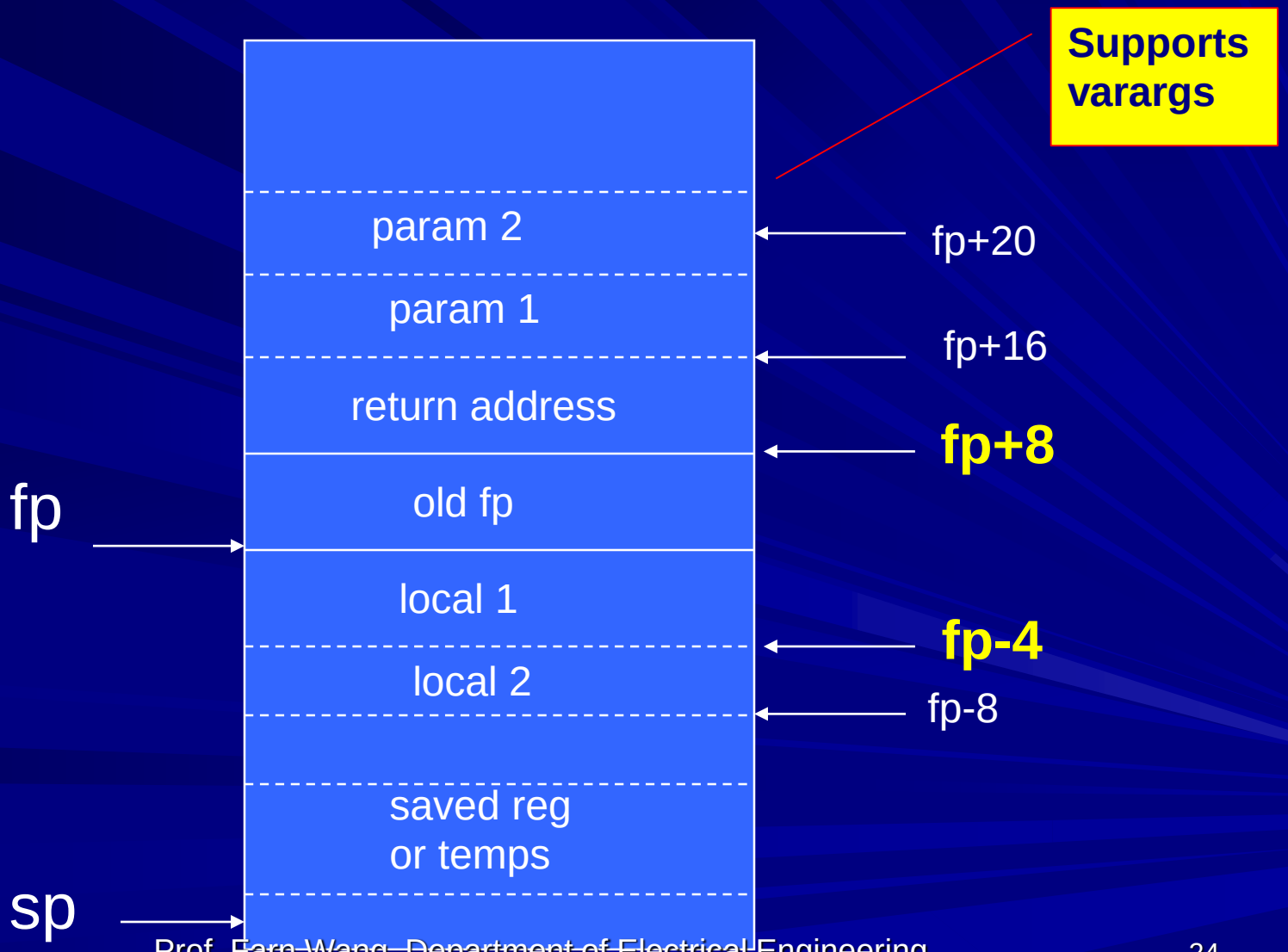
if (array) print (var name, “:”, “.space”, size);

if (initialized int) print(var name,”:”, “.word”, initial_value);

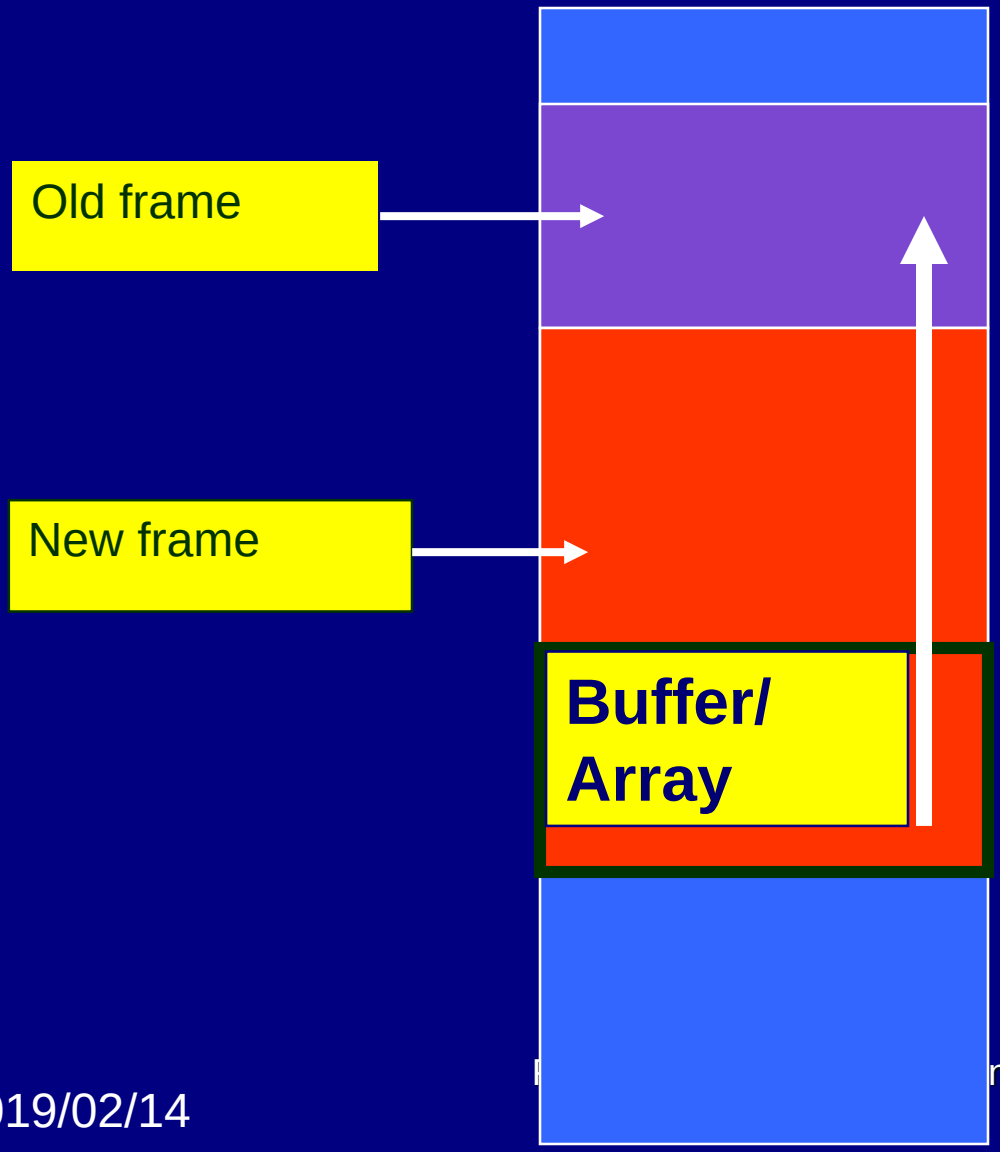
if (initialized float) print(var name,”:”, “.float”, initial_value);

if (local) tracking offset, enter offset into the symbol table.

Stack allocation: AR layout



How virus/worms get implanted?



When buffer overflows, it can write into the bookkeeping infor in the previous frame. e.g. it could change the return address

Advantages of using FP

- Why do we need both fp and sp? Would sp be sufficient for addressing all objects on the frame?

  e.g. gcc has an omit-frame-pointer option
- The frame-pointer (fp) provides a fixed reference into the stack frame. All locals are (fp) – offset. Arguments are referenced with (fp) + offset, the leftmost argument will always be (fp)+16, return address always (fp)+8.
- For modern PLs, which support variable length allocation on stack, using **fp** is a better approach for frame base than using SP.

Advantages

- In One-pass compilation, the offset from SP is not easy to calculate since the AR size is not known until the complete procedure is compiled.
- Using frame pointer (fp) can avoid back-patching.

Example

What is the offset relative to (fp) for each reference?

1) b — fp + 20
2) j — fp - 8
3) Return address — fp + 8
4) e — fp - 56

```
functionx (int a, float b) {
    int i,j,k;
    float d[10], e;
    .....
}
```

Example

```
Functionx (int a, float b) {
    int i,j,k;
    float d[10], e;
    …..}
```

What is the offset relative to (fp) for each reference?

1) d[0] — fp - 52
2) d[9] — fp - 16
3) d[10] — fp – 12
4) d[15] — fp + 8 (return addr)

Offset tracking and alignment

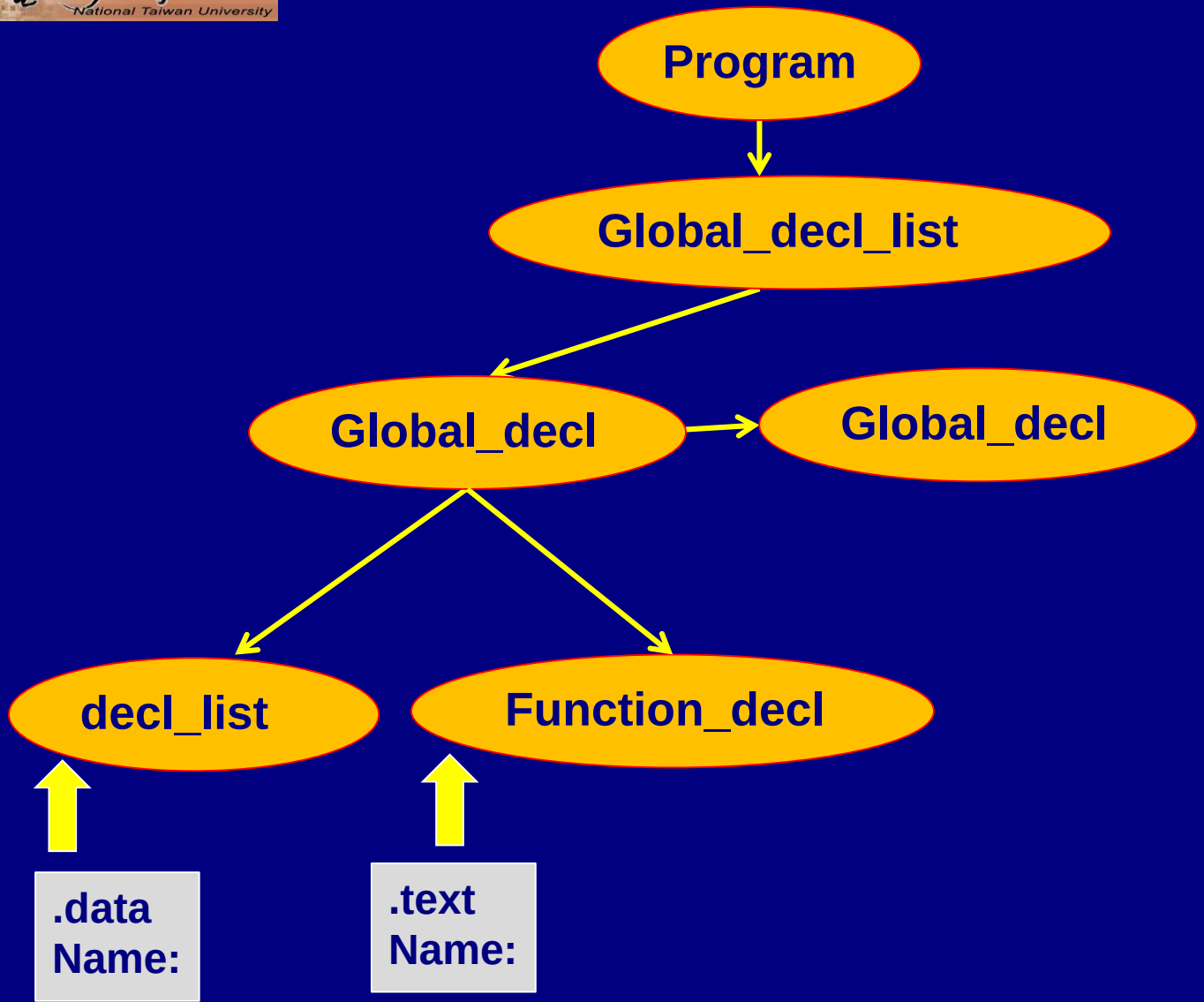
- function_decl : type ID ‘(‘ param_list ‘)’

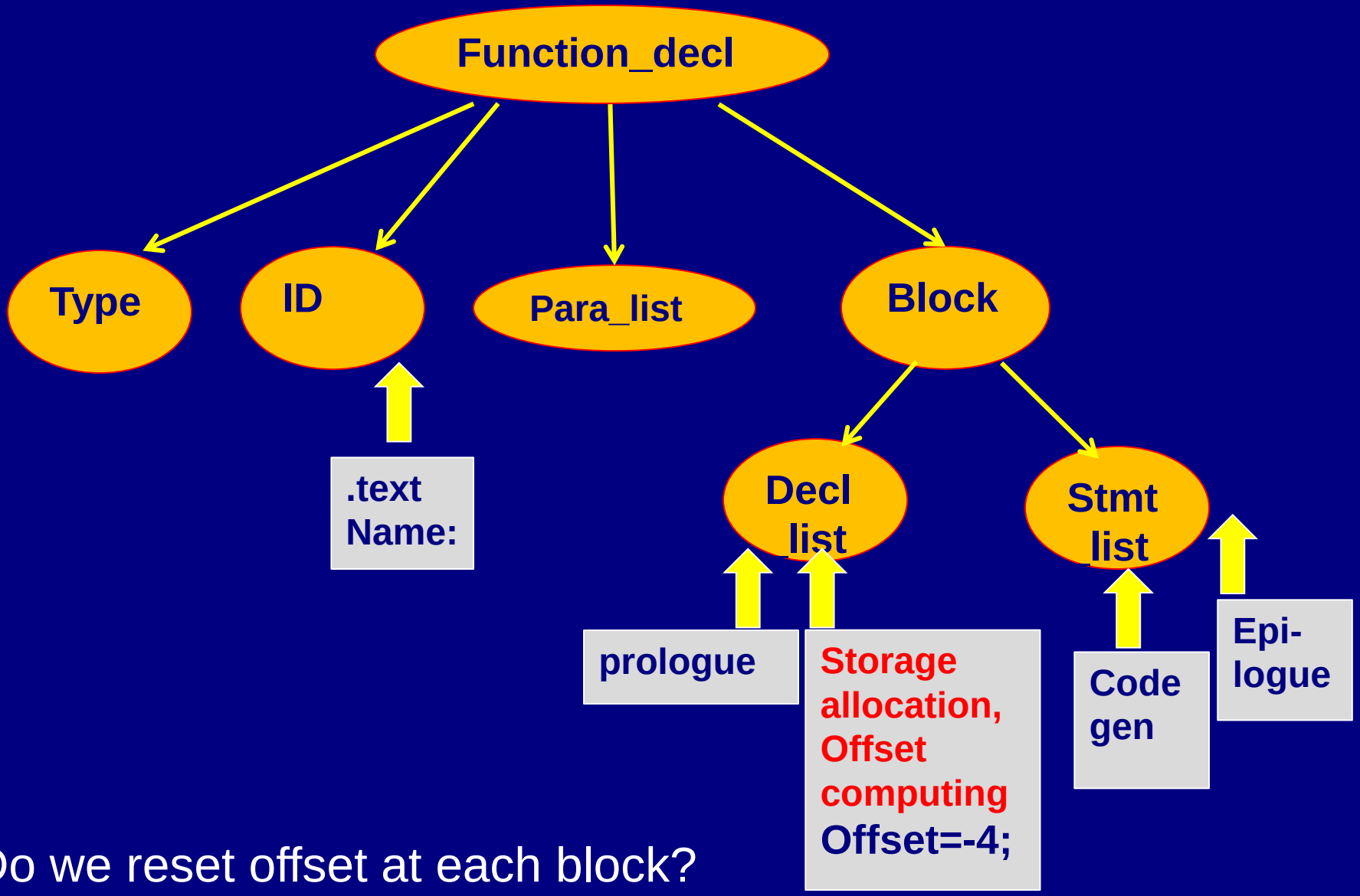
  initialization: offset = 4;
- In id_list, init_id_list, when ID is processed

  offset = offset + get_sizeof (id’s type)
- if alignment is needed (not in this project)

  align_size = get_alignment(id’s type);

  while (offset % align_size) ++offset;
- Do we reset offset at each block? Or only once at the beginning of the function main block?

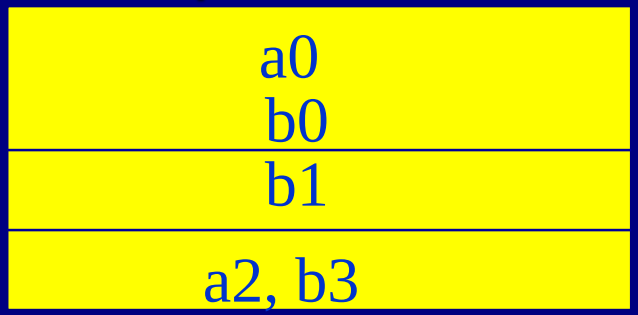

Program
Global_decl_list
Global_decl
Global_decl
decl_list
Function_decl
.data
Name:
.text
Name:



Do we reset offset at each block?
Or only at the beginning of the function main block?

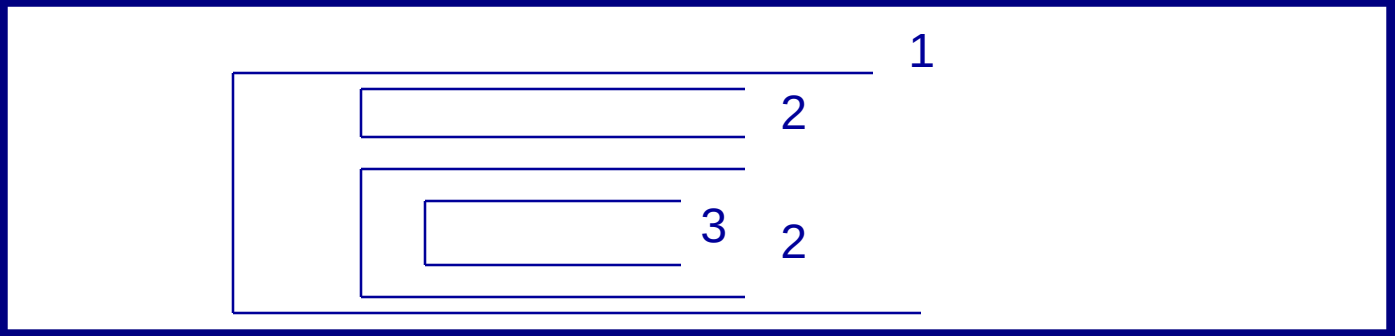
Block-level and Procedure-level ARs

- Each block can be considered as an in-line procedure without parameters. So we could create a new AR for each block. This is block-level AR and is very inefficient.
- In procedure-level AR method, AR is only used for a true procedure. The relative location of variables in individual blocks within a procedure can be computed and fixed at compile time.



Procedure level AR layout

- What if there are many nested blocks in the function? Locals of blocks at the same nesting level could share the same storage area.



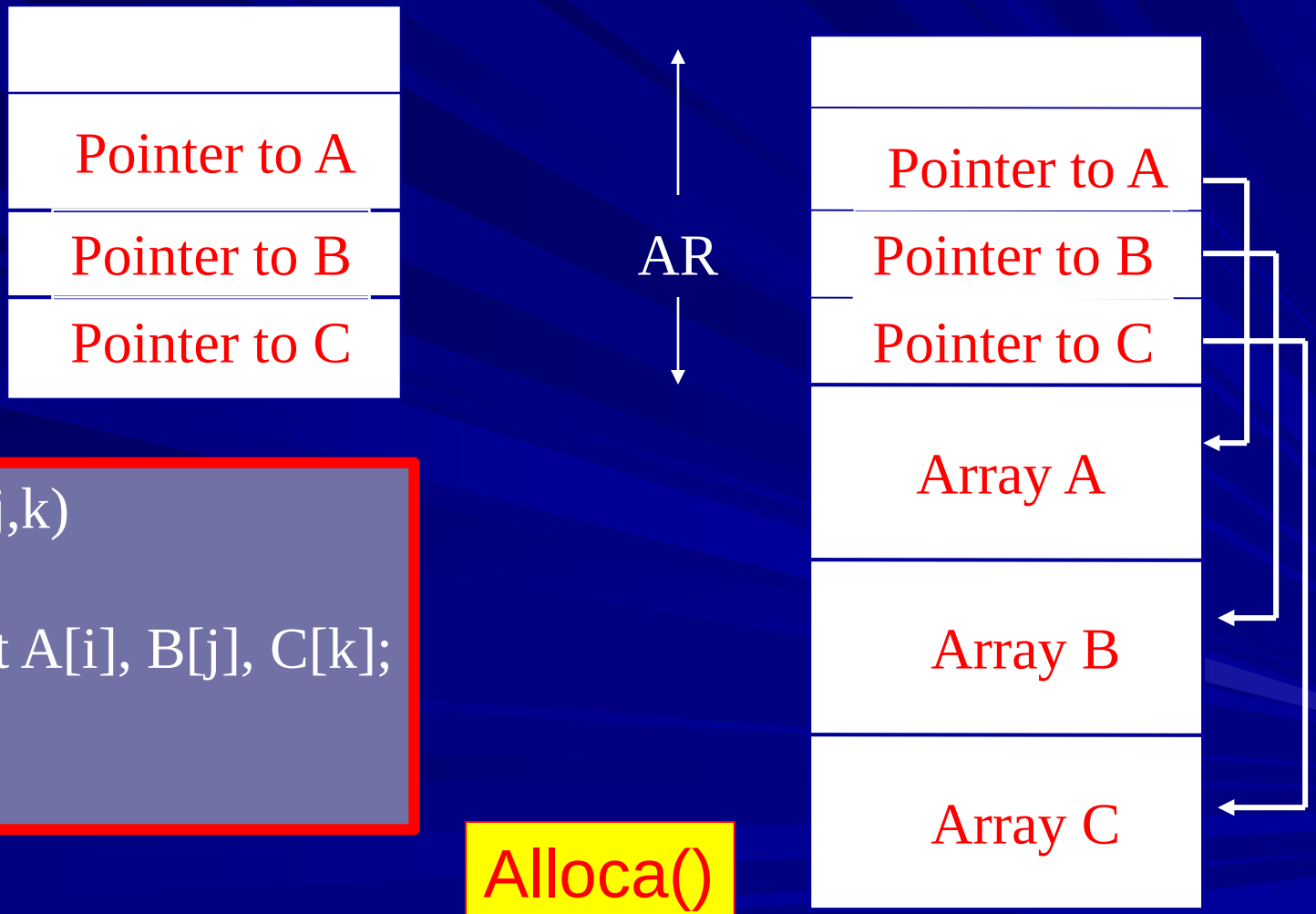
- Need an array or a stack to remember the accumulated offset at each nesting level.
- The largest offset is remembered as the size of the local variable area in the procedure level AR.

Dynamic Arrays

```
f(int n) {
    float a[n], b[n*10];
    …
}
```

- The bounds of dynamic arrays are determined at runtime rather than at compile time.
- Dynamic arrays cannot be allocated within an AR, they can be allocated on the top of stack as soon as their associated declaration is evaluated.

Dynamic Arrays

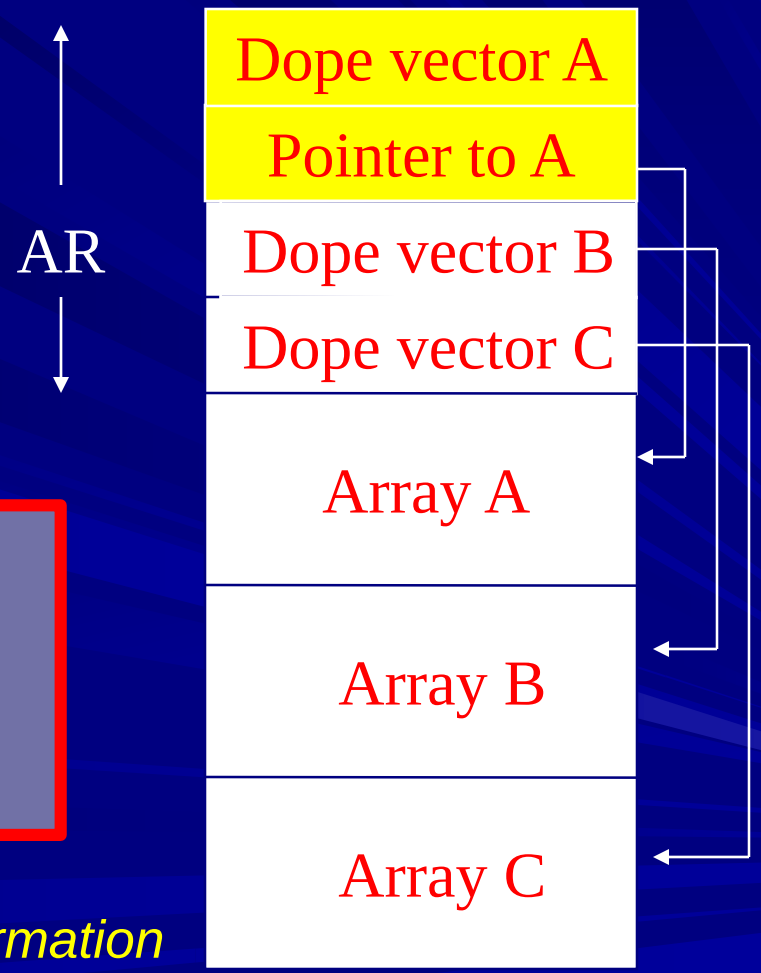



```
f(int i,j,k)
{
    float A[i], B[j], C[k];
    ….
}
```

Alloca()

Multi-Dimensional Dynamic Arrays: Dope vectors

- Dope vector is a fixed size descriptor containing the array's type declaration.
- It contains the dimension, size, and bounds info of the array.




```
f(int i,j,k) {
    float A[i][j], B[j][k], C[i][j][k];
    ….
}
```

***Dope**: facts, private or confidential information*
Slang for drugs -☺

Calling Sequences

- A call sequence allocates an activation record and enters information into its fields.
  - parameters, return address, old sp/fp, saving registers, local data initialization
- A return sequence restores the state of the machine
  - return value, restore registers, restore old sp/fp, branch to return address
- The code in a calling sequence is often divided between the caller and the callee.

Calling Sequences

- A possible calling sequence
  - Caller evaluates actual parameters
  - Caller stores a return address and the old stack top (or frame pointer) into callee's AR.
  - Caller pushes the AR (i.e. decrement sp)
  - Callee saves register values and other status information
  - Callee initializes local data and begin execution

Calling Sequences

- Calling sequence used in our template
  - Caller evaluates actual parameters
  - Caller stores a return address
  - Callee pushes AR
  - Callee stores the old stack top and the frame pointer into AR.
  - Callee saves registers and other status information
  - Callee initializes local data and begin execution

Return Sequences

- A possible return sequence
  - Callee places a return value next to the AR of the caller.
  - Callee restores old fp/sp and other registers
  - Callee branches to the return address

  - Caller copies return value into its own AR

Heap Deallocation

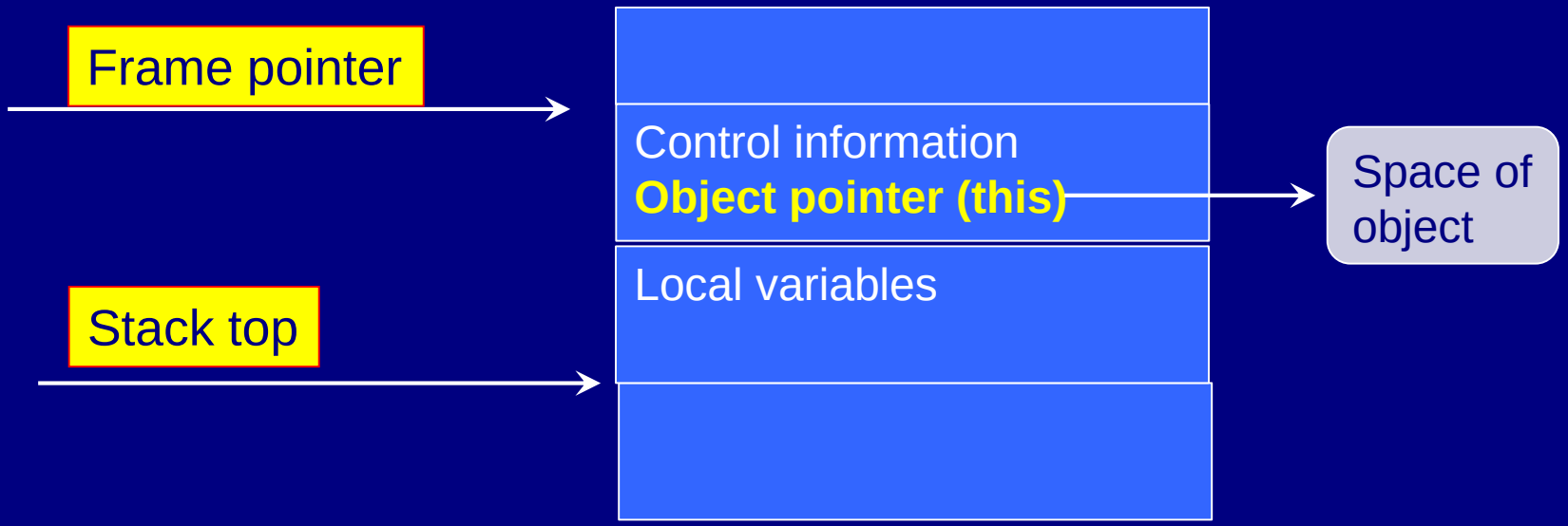
- No deallocation
  - Stop when space runs out
- Explicit deallocation
  - free (C, PL/1), dispose (Pascal), deallocation (Ada)
  - May lead to dangling references
- Implicit de-allocation
  - Reference count
  - Garbage collection
    - Mark and Sweep

Overhead
Circular heap

Multiple Scopes

Classes and Objects

- Java, C++ and C# allows classes to have member functions that have direct access to all instance variables.
- When the member function executes, it requires two pointers: frame pointer for locals, and object pointer for the instance variable.



Multiple Scopes

Classes and Objects

Non-local Names

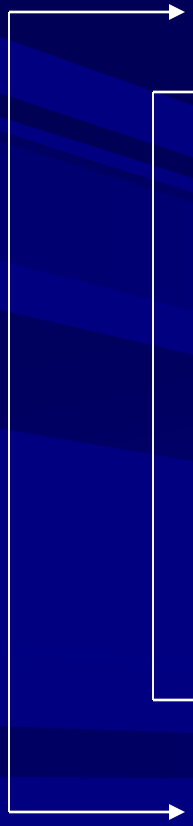
- Ada, Pascal, Algol 60, Python, ML and Scheme allow subprogram declarations to nest.
- This introduces the needs to access non-local names on the stack.
- Subprogram nesting is useful, allowing, for example, a private utility procedure to access another routine's locals and parameters

Non-local names

- In a language with nested procedures (or blocks) and static scope (lexical scope), some names are neither local nor global, they are non-local names.

```
procedure A
real a;
    procedure B
    real b;
      reference a;  ← non-local reference
    end B
end A;
```

Example: Non-local names in C

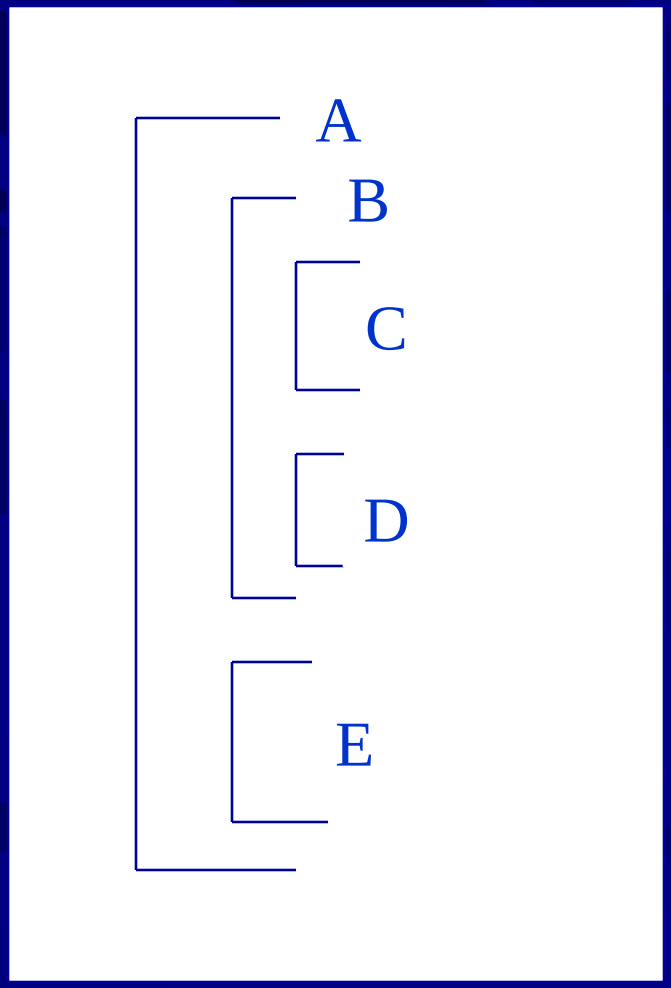


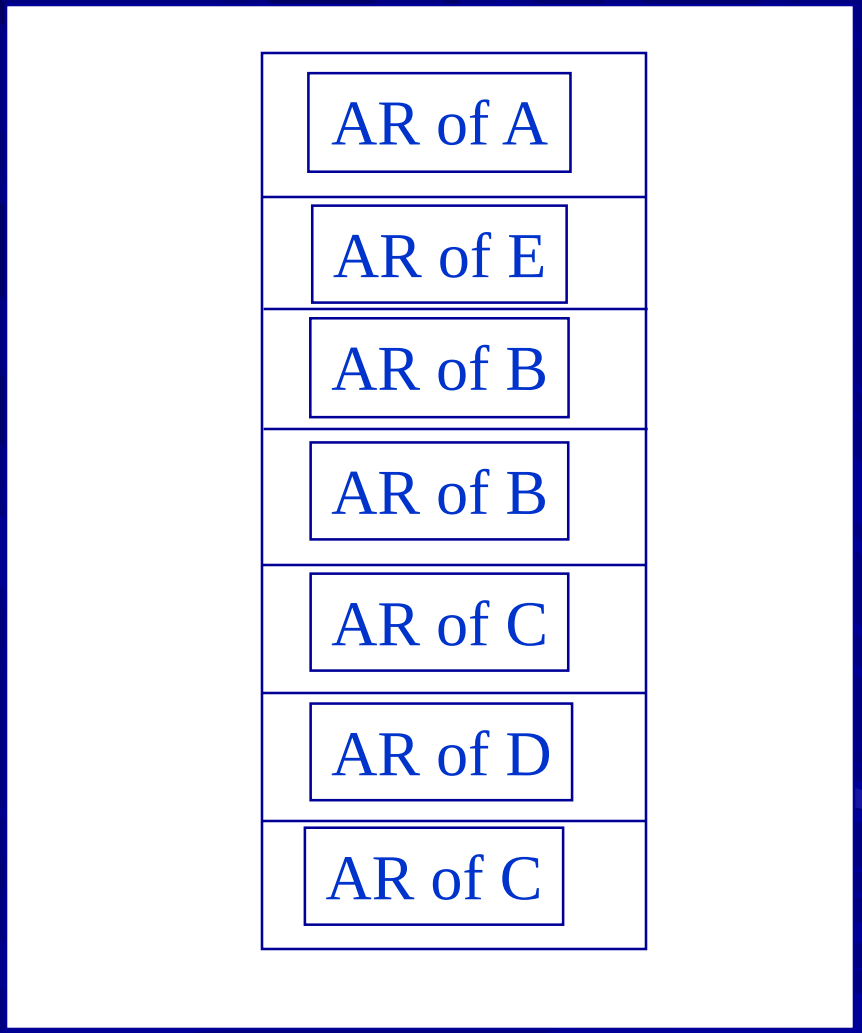
```
main () {
int a = 0, b=0; {
     int b = 1;     {
                    int a = 2;
                    print(a,b); }     2, 1
                    {
                    int b = 3;
                    print(a,b); }     0, 3
     print(a,b);}   0, 1
print(a,b); }  0, 0
```

Most closely nested rule

Example of non-local references

Duplication of AR for non-locals ?

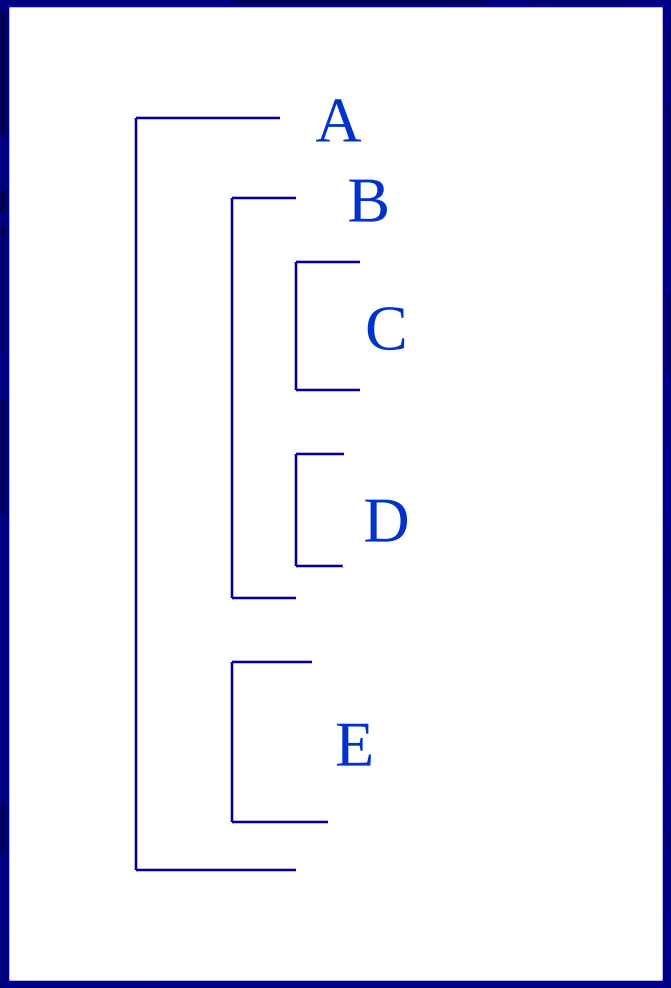


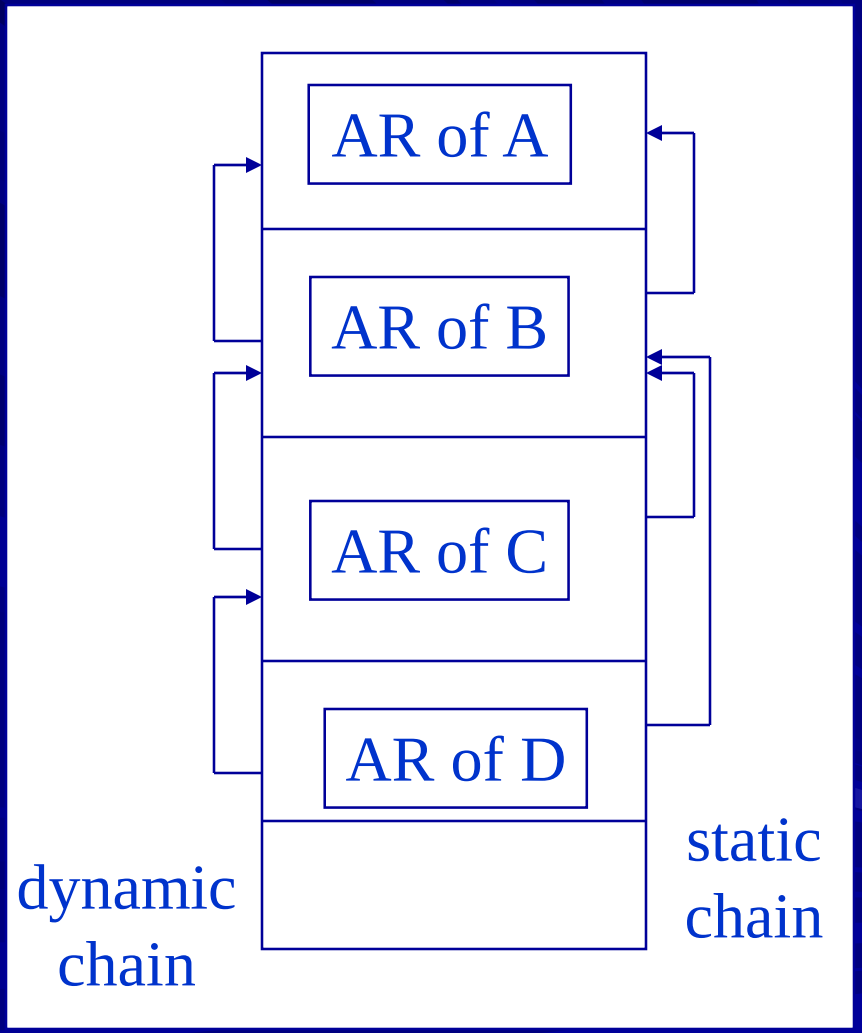


Static/Dynamic Chains

- Static link: each stack frame contains a reference to the frame of the lexically surrounding procedure.
  - By following the static links (a static chain), the non-local object can be found.
  - Static link is also called access link.
- Dynamic link: The saved value of the frame pointer, which points to the caller's AR, is a dynamic link.
- Non-local references by static chain are slow.

Example of Static Chain





Passing Static Link

- When A calls B, the caller, A, needs to pass a static link to the callee, B.

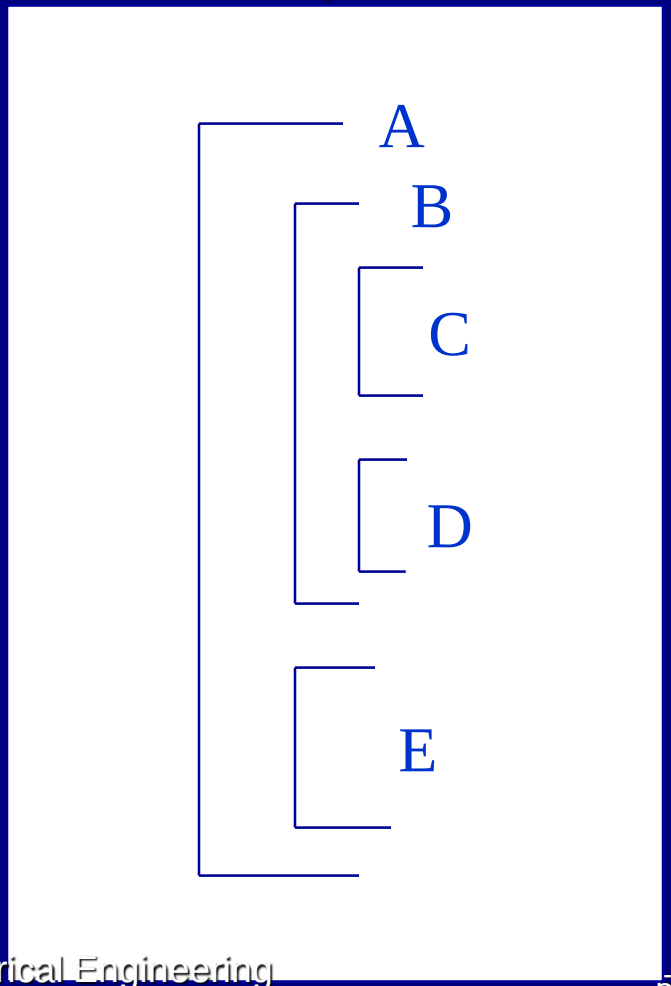
  Three possible calling cases:

  - Parents call children

    e.g. A calls B, B calls C
  - Call at the same level

    e.g. C calls D, B calls E
  - Calling ancestors

    e.g. D calls B, C calls E



Passing Static Link

Three possible calling cases:

- Parents call children

  e.g. A calls B, B calls C

  Caller passes its AR pointer

- Call at the same level

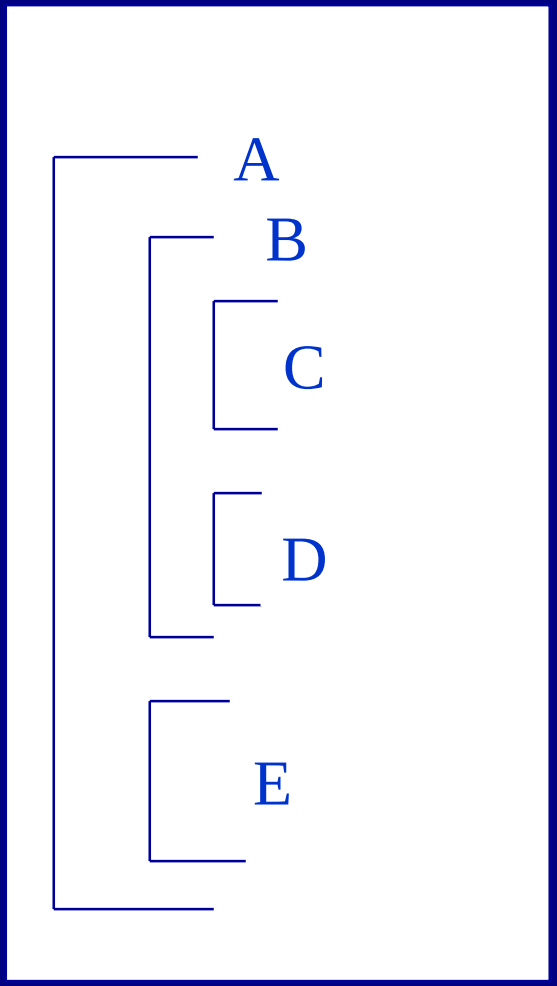
  e.g. C calls D, B calls E

  Caller passes its static link

- Calling ancestors

  e.g. D calls B, C calls E

  Caller traverses static links to locate the correct ancestor's AR



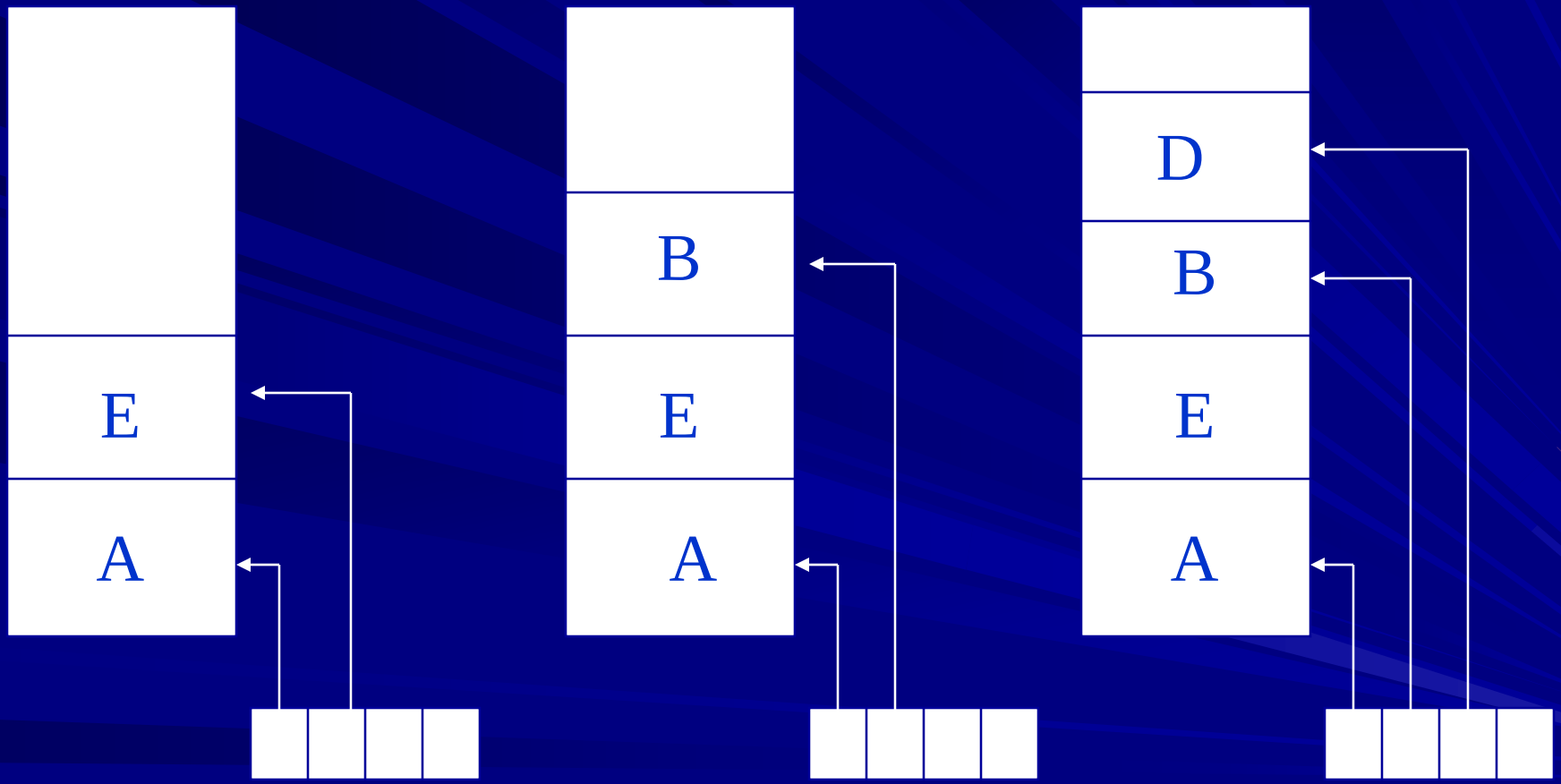
e.g. C (level 3) calls E (level 2), C needs to follow its static once to obtain A's AR.

Displays

an embedding of the static chains into an array.

- The *jth* element of the display contains a reference to the frame of the most recently active procedure at lexical nesting level j.
- can be stored in a set of registers, or in memory.
  - Non-local names are not referenced very frequently, so keeping the display in registers may not be as efficient as it originally thought

Example of Display



Closures and Cactus Stack

- Closures

When functions are *first-class objects* (*they can be stored in variables and data structures, constructed during runtime and returned as function results*), when such functions are created or manipulated, we must maintain two pointers: one to the code, and one to its frame. This pair of pointers is called a **closure**.

- Cactus Stack

Concurrent execution of multiple lightweight threads requires each thread has its own stack. Since these threads might share the main stack, a cactus stack structure is often created. (saguaro stack)

Parameter Passing

- Parameters
  - Names that appear in the declaration of a procedure are formal parameters.
  - Variables and expressions that are passed to a procedure are actual parameters (or arguments)
- Parameter passing modes
  - Call by value
  - Call by reference
  - Copy-restore
  - Call by name

Call-by-Value

- The actual parameters are evaluated and their *r-values* are passed to the called procedure.
- A procedure called by value can affect its caller either through nonlocal names or through pointers.
- Parameters in C are always passed by value. Array is unusual, what is passed by value is a pointer.
- Pascal uses pass by value by default, but *var* parameters are passed by reference.

Call-by-Reference

- Also known as call-by-address or call-by-location. The caller passes to the called procedure the *l-value* of the parameter.
- If the parameter is an expression, then the expression is evaluated in a new location, and the address of the new location is passed.
- Parameters in Fortran are passed by reference

an old implementation bug in Fortran

```
func(a,b) { a = b};
call func(3,4); print(3);
```

Copy-Restore

- A hybrid between call-by-value and call-by reference.
- The actual parameters are evaluated and their R-values are passed as in call-by-value. In addition, L-values are determined before the call.
- When control returns, the current R-values of the formal parameters are copied back into the L-values of the actual parameters.

Why Copy-Restore

- Copy-Restore avoids the problem when a procedure call has more than one way to access a variable.

```
program A
int a;
    program f(var x)
    {    x:=2; a:=0; }
{ a:=1;  f(a);  print(a) }
```

- In the above example, call-by-reference will print 0, copy-restore will print 2.
- Avoid potential problems with alignment. Small objects may not be aligned if they are packed in a record.

Call-by-Name

- The actual parameters literally substituted for the formals. This is like a macro-expansion or in-line expansion.
- Call-by-name is not used in practice. However, the conceptually related technique of in-line expansion is commonly used.
- In-lining may be one of the most effective optimization transformations if they are guided by execution profiles.

Summary

- Binding names to storage locations
- Activation Record (AR)
- Storage allocations
- Accessing non-local names
- Parameter passing