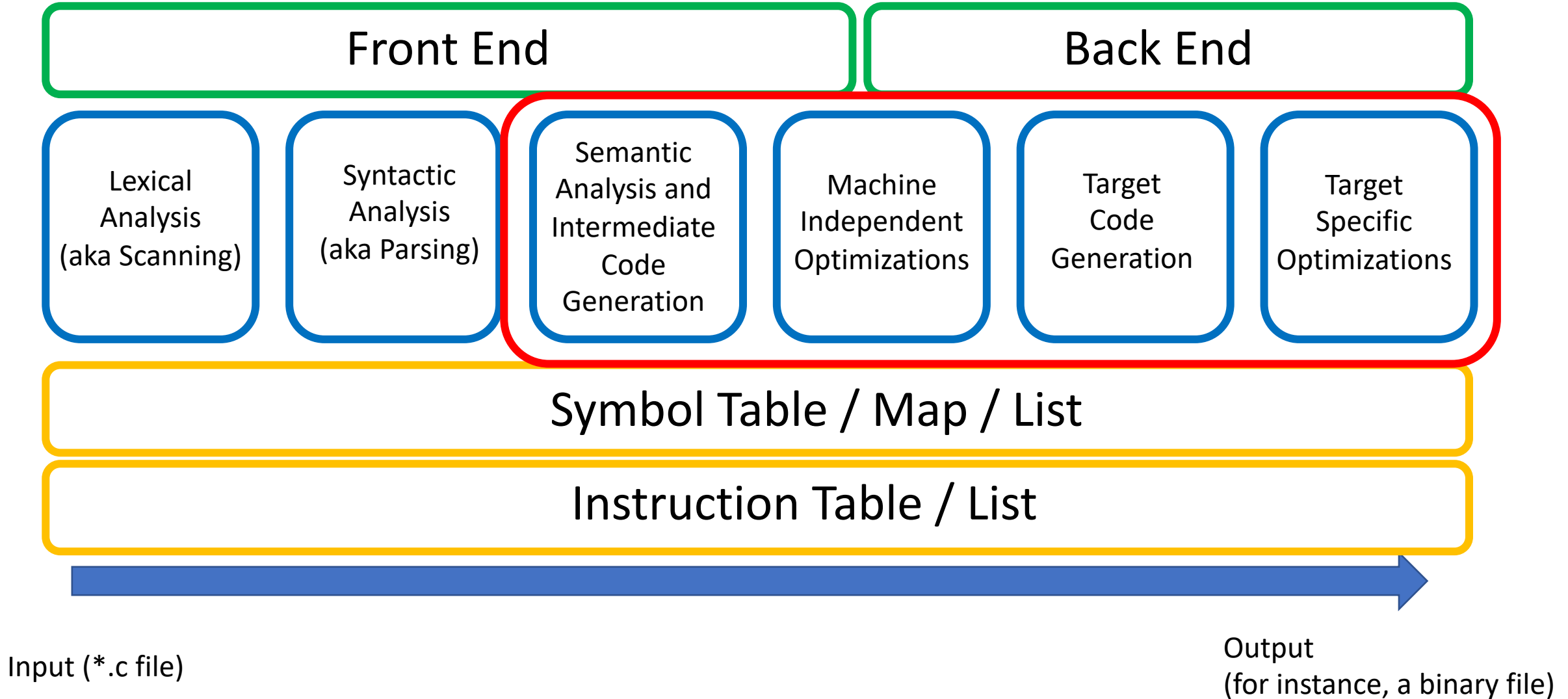


The Environment: Names, Scopes and Bindings

Overview

- Binding time
- Object lifetime and Storage Management
 - Program's stack: static memory allocation
 - Program's heap: dynamic memory allocation
- Scope rules:
 - "what variable (memory location) am I using in location X?"
 - Reach of a declaration
 - Visibility of variables, functions, objects, methods, etc.

Compilation Overview



The Need for Abstractions

- Abstraction:
 - brings something to a higher-level (Compare assembly code against Java code)
 - Hides irrelevant details
 - Focuses on main properties
- Names / symbolic identifiers: addresses and computations
- Subroutines:
 - control abstractions
 - Jump from one point in the program to another one
 - Records / remembers previous state
 - Defines how to pass arguments
 - Defines how to return results

The Need for Abstractions

- Classes (in the Object-Oriented sense):
 - Data abstractions
 - Group data and computations together
 - Abstract “security”

Binding Time

- Binding: association between two things
- Example:
 - name and a storage location
 - name and implementation of a subroutine
- Binding time: the time at which a binding is made
- Several binding times:
 - Language design time: control-flow constructs, data types
 - Language implementation time: some aspects left as “implementation specific”, e.g. the number of bits used in floating point precision
 - Program writing time
 - Compile time: mapping of PL constructs to machine code and memory layout
 - Link time: functions in separate compilation units
 - Load time: when the program is loaded by the operating system; logical to physical address translation
 - Run time: a function activation, dynamic memory allocation

Binding Time

- Also: static vs dynamic
- Umbrella term for several binding times in previous slide
- Compiler-based language implementations tend to be more efficient:
 - Decide layout / location for variables
 - Generates more efficient code
 - Particularly useful in loop-based coded (hotspots in programs)
 - Some decisions are "local best": addresses of variables
- Interpreted languages:
 - Decisions and optimization are time constrained
 - Vital information might not be available yet at compile time
 - Most scripting languages delay type-checking to runtime

Object Lifetime

Key events:

- Creation and destruction of objects
- Creation and destruction of bindings
- Deactivation and reactivation of bindings that may be temporarily unusable
- Reference to variables, subroutines

Lifetime

- Binding lifetime: time between creation and destruction of name-to-object binding
- Object lifetime: time between creation and destruction of an object

```
int a;  
while (a < 10)  
{  
    int i;  
    ...  
    if ( a % 2 == 0) {  
        int i;  
        ...  
    }  
}
```

Lifetime

- Binding and object lifetime do not necessarily coincide
- Object may retain value and access potential even without name (binding):
 - Pass by reference & in C++: lifetime of binding shorter than lifetime of object
- Also possible to have a binding lifetime longer than an object lifetime (likely sign of bug):
 - Dangling reference: a pointer without memory associated to it (or memory freed)

Object Lifetimes

- Static: objects with absolute address throughout the program execution
- Stack: objects are allocated and deallocated in LIFO order (Last In, First Out)
 - Example: regular and recursive function calls
- Heap: may be allocated and deallocated at arbitrary times during the program execution; require more general and expensive (timewise) storage management

Note: You probably have heard of these 3 memory segments in your OS or Computer Architecture / Organization class

Static Allocation

Several examples:

- Global variables
- Program instructions
- Variables that retain value, but associated to single subroutine (e.g. C static variables)
- Numeric and string constant literal, e.g. 10, 10.0 and “10”
- Compiler tables and data structures used in debugging, garbage collection, exception handling

Static Allocation

- Local variables (to subroutines) are created when the subroutine is called and destroyed when it ends
- Each subroutine call creates a new, separate instance of each local variable
- However, not all languages do (or used to do) this by design:
 - Fortran did not originally support recursion (direct or indirect)
 - Added in Fortran 90
 - Implication 1: could not have 2 or more active calls to the same subroutine
 - Implication 2: essentially no distinction between global and stack variables, but subroutines still called in LIFO order

Static Allocation

- Interesting piece of info in book (page 119):
 - Design decision influenced by cost of manipulating stack in IBM 704 (introduced in 1954)
 - Programmers had to wait ca. to 30 years for recursive subroutine support
- Compile time constants defined in subroutines can be statically stored:

```
int f () {  
    const char * mystr = "error message";  
    // will never change  
    ...  
}
```

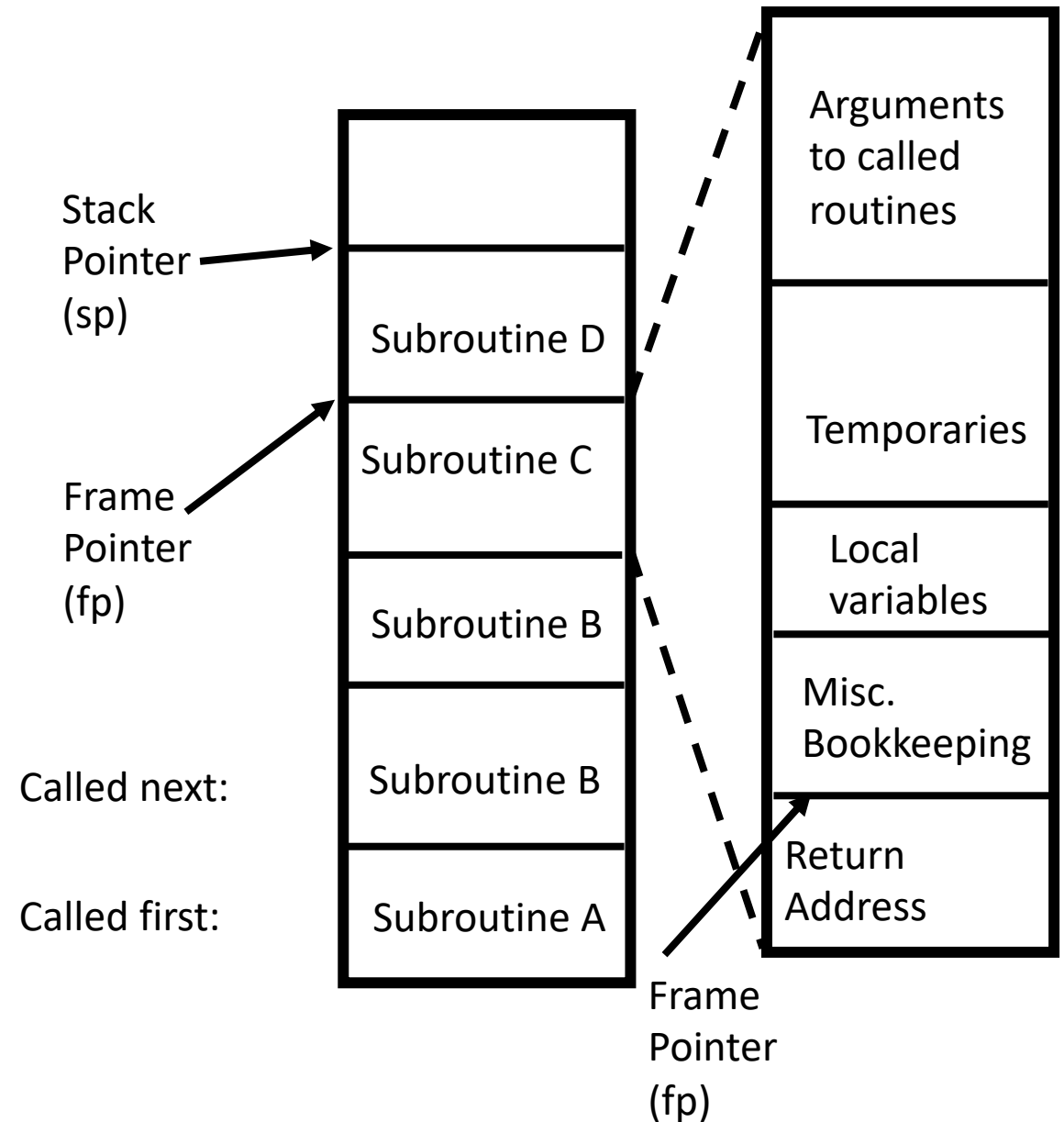
Static Allocation

- Compilers can allocate memory for a single instance of the constant, and allow all the calls of a subroutine to use it.
- In other languages (e.g., C and Ada) the compiler initializes the constant at runtime, as it could depend on some other variables:
 - Constants then allocated on stack
 - C# distinguishes between compile-time and elaboration-time constants with the keywords `const` and `readonly`, respectively.
- Elaboration time = initialization time

```
void f (int a) {  
    const int b = 3;  
    const int c = a + 1;  
    printf ("%d %d %d", a, b, c);  
    if (a > 0)  
        f (a - 1);  
}
```

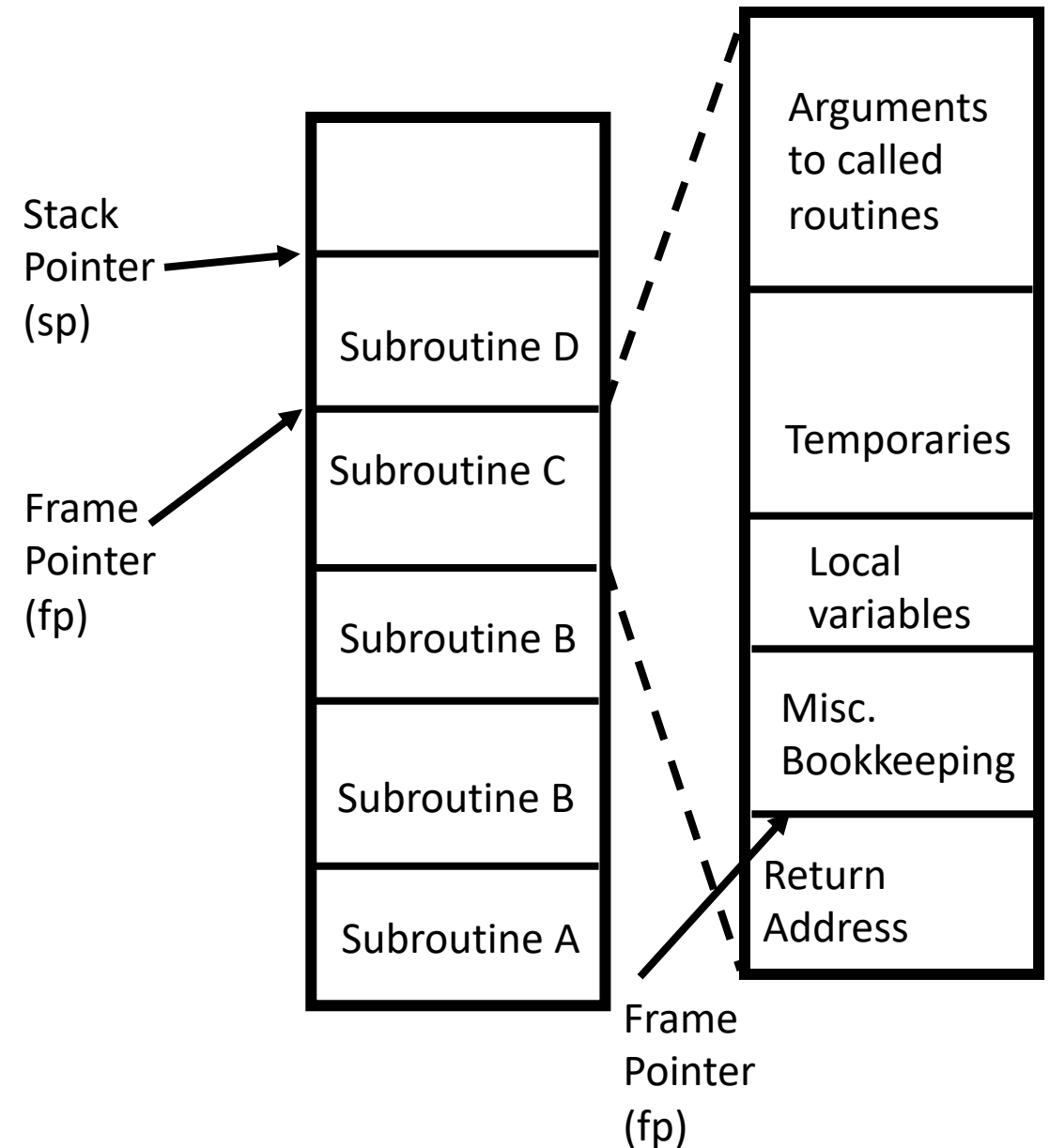
Stack-based Allocation

- Recursion complicates static allocation of variables
- Number of instances of a variable (e.g., a variable named “count” in a function “sum”) is, in theory, unbounded
- Natural nesting of functions allows to allocate memory on the stack
- Each instance (call) of a subroutine assigns memory from the stack for the various variables and constants used in the subroutine



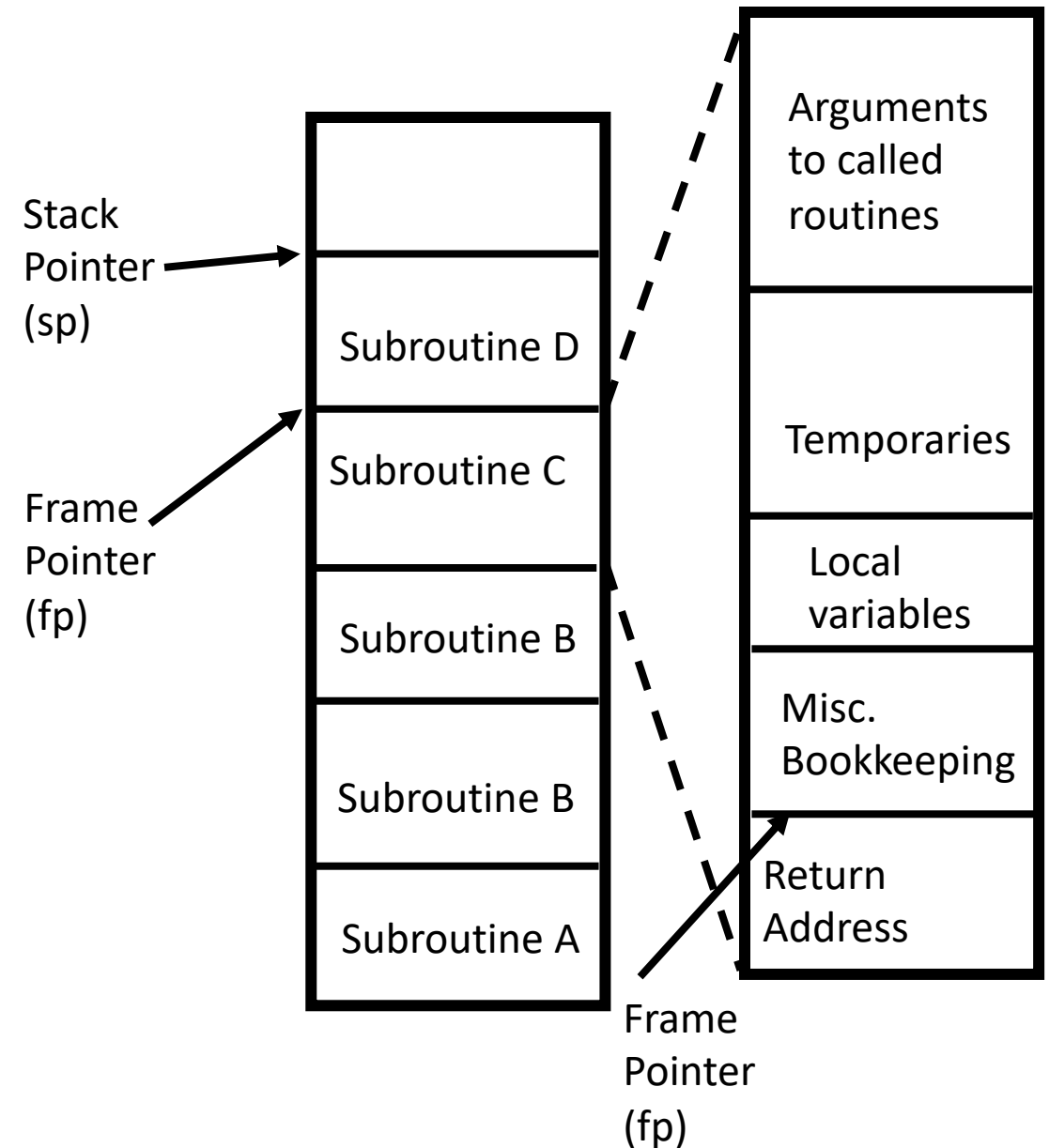
Stack-based Allocation

- Memory of subroutine allocated in a frame or activation record
- Frame also allocated memory for temporary variables (produced by compiler)
- Bookkeeping information includes: return address, reference to frame of caller (dynamic link), saved values of registers needed by caller and callee (e.g., the program counter)



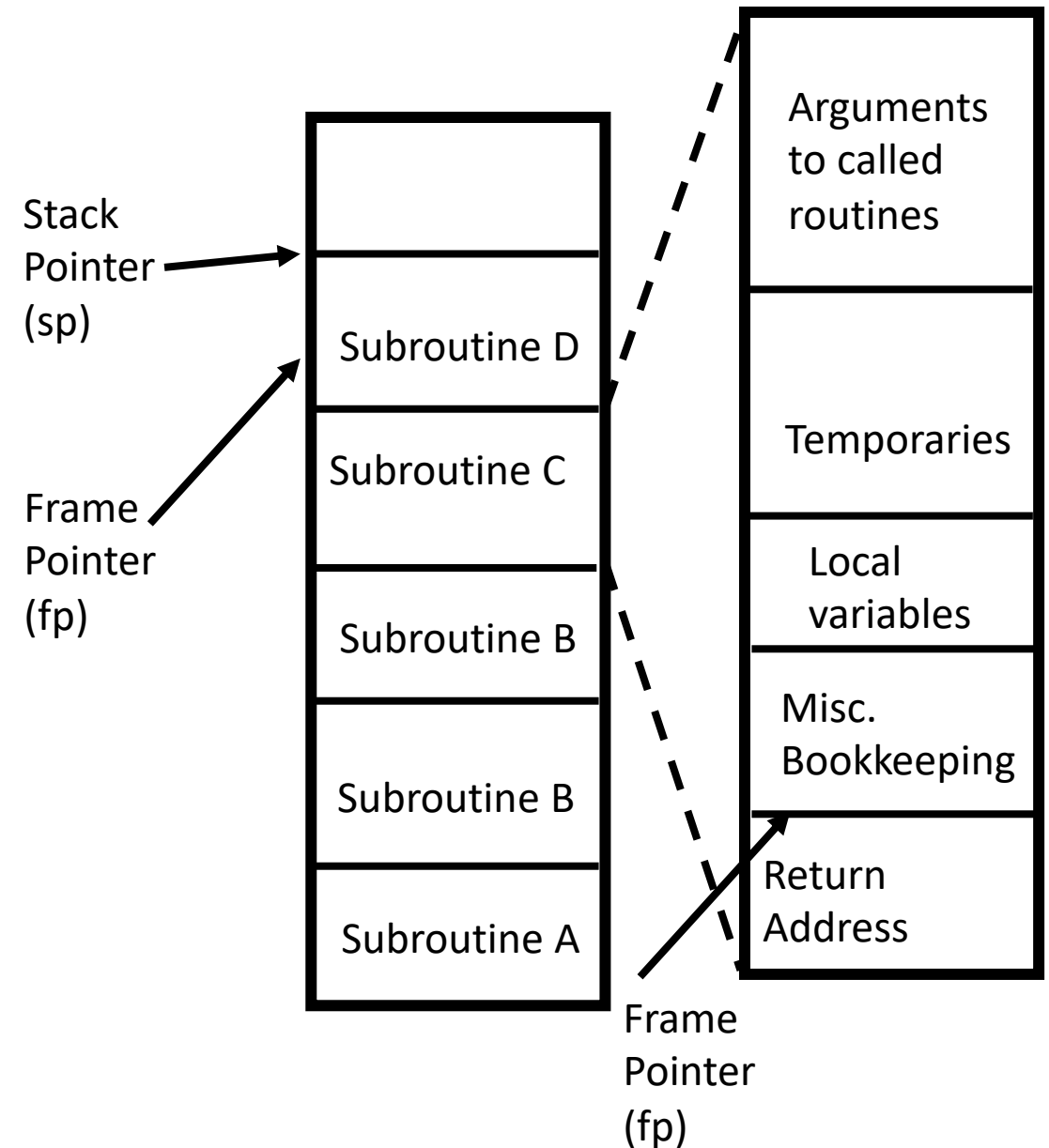
Stack-based Allocation

- Arguments passed to subroutines lie at the top of the frame (it's much more convenient)
- Subroutine actual arguments usually pushed into the stack
- Memory layout very implementation and language dependent
- Stack maintenance responsibility of the caller, before and after calling sequence
- Two parts: prologue and epilogue



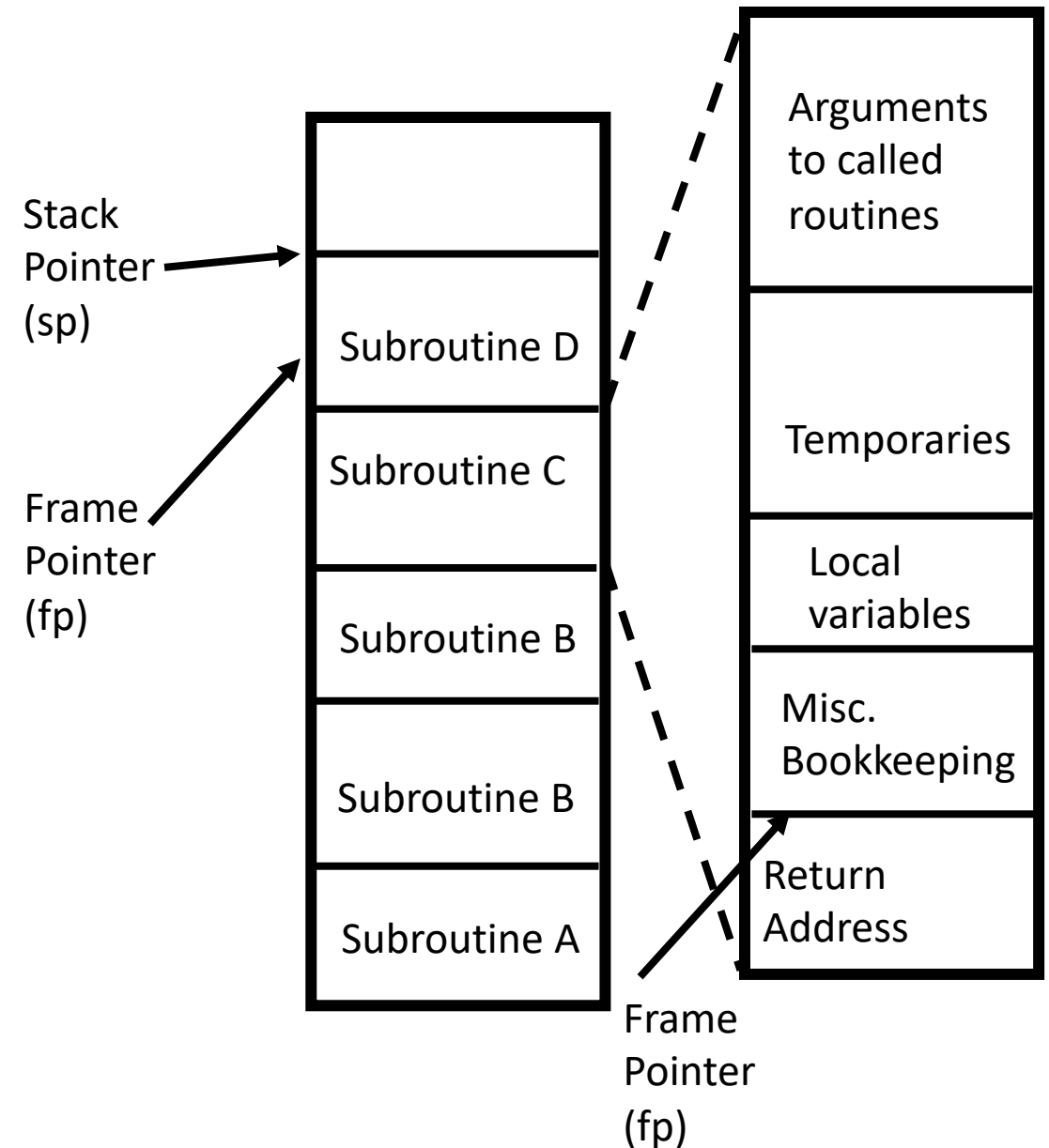
Stack-based Allocation

- Location (actual address) of stack frame not determinable at compile-time, but offsets within frame are
- Frame pointer (fp) points to known location within the new activation frame; useful for external access such as copying results
- Other addresses accessed by known offsets w.r.t. fp
- FP, SP, PC usually correspond to machine registers



Stack-based Allocation

- Offset of variables (w.r.t fp) can be used in *load* and *store* instruction variants
- Offsets computed statically by compiler with datatype information (e.g. a char is 1 byte, int usually 4 bytes, double usually 8 bytes, struct sum of fields)
- Stack grows downward, towards lower addresses in most language implementations
- Machine instruction sets usually will provide push and pop instructions for frame manipulation
- Offsets will normally be negative



Maintaining the Static Link

- Need some additional work in languages with nested subroutines
- Most work falls in the caller side
- Work depends in the nesting of the callee:
 - Callee nested inside caller: caller passes itself as the static link of the callee
 - Callee is $k \geq 0$ scopes outward of caller:
 - Scopes surrounding callee also surround caller
 - Caller dereferences its own static link k times and passes the result as the static link of the callee

Typical Calling Sequence

Caller

- Save registers
- Compute values of arguments, move them into stack / registers
- (if language with nested subroutines) Compute static link and pass it as a hidden argument
- Use special instruction to jump to address start of subroutine (includes saving the return address in the stack or in a register)
- Moves returned values to wherever needed
- Restores pending registers

Callee

- Allocates a frame by deducting some constant offset from the sp
- Saves old frame pointer into the stack, update to the newly allocated frame
- Move return value (if any) to a register, or specific address given by the caller (possibly in the stack)
- Restores register values from the caller (inc. fp and sp)
- Jumps to return address

Prologue

Epilogue

Parameters in Programming Languages

Parameter mode	Representative languages	Implementation mechanism	Permissible operations	Change to actual?	Alias?
value	C/C++, Pascal, Java/C# (value types)	value	read, write	no	no
in, const	Ada, C/C++, Modula-3	value or reference	read only	no	maybe
out	Ada	value or reference	write only	yes	maybe
value/result	Algol W	value	read, write	yes	no
var, ref	Fortran, Pascal, C++	reference	read, write	yes	yes
sharing	Lisp/Scheme, ML, Java/C# (reference types)	value or reference	read, write	yes	yes
r-value ref	C++11	reference	read, write	yes*	no*
in out	Ada, Swift	value or reference	read, write	yes	maybe
name	Algol 60, Simula	closure (thunk)	read, write	yes	yes
need	Haskell, R	closure (thunk) with memoization	read, write [†]	yes [†]	yes [†]

Figure 9.3 Parameter-passing modes. Column 1 indicates common names for modes. Column 2 indicates prominent languages that use the modes, or that introduced them. Column 3 indicates implementation via passing of values, references, or closures. Column 4 indicates whether the callee can read or write the formal parameter. Column 5 indicates whether changes to the formal parameter affect the actual parameter. Column 6 indicates whether changes to the formal or actual parameter, during the execution of the subroutine, may be visible through the other. *Behavior is undefined if the program attempts to use an r-value argument after the call. [†]Changes to arguments passed by need in R will happen only on the first use; changes in Haskell are not permitted.

Heap-based Allocation

- Heap: program segment
- Program can use memory from it during its execution
- Require two abstractions: allocation (e.g. `new`) and deallocation (e.g. `free`)
- BTW: Dynamic allocation is a performance killer
- Heap management was an active research area: speed and space concerns
- Space: internal (unused within a block) and external fragmentation (used and unused blocks/space scattered and interleaved)

Heap-based Allocation

- Very often, memory blocks kept in a single linked list: the free list
- At program start, free list contains a single node
- Each new memory request assigns slots from the free list → free list grows
- Several algorithms:
 - First first: select the first slot in the free list with sufficient capacity
 - Best fit: traverse the whole free list to find the best match → higher allocation cost
- If chosen block is too large, it's partitioned; unneeded part returned to the free list
- Options for coalescing when the memory is returned to the *pool* (the heap)

Heap-based Allocation

- Single free list incurs in linear cost in the number of free blocks
- Alternative: have free lists of various sizes
- Requests rounded up to next standard size
- Sizes can be set statically or dynamically
- Issues with external fragmentation:
 - Degrade performance over time
 - Make harder to satisfy new memory requests
- Two common mechanisms:
 - Buddy system: powers of 2
 - Fibonacci heaps: slightly lower internal fragmentation (sequence grows slower than powers of 2)
- Compaction eliminates external fragmentation: find and move already allocated blocks

Garbage Collection

- Allocation and deallocation of objects is explicit in some languages (malloc, new and free operations in C and C++)
- In other languages, objects (broadly speaking) are automatically deallocated by the runtime (aka the environment)
- Garbage collector: part of the runtime in charge of finding and freeing objects that are not used anymore
- Examples of languages with GC: Java, C#
- Arguments in favor of explicit object management: speed and simplicity (shift burden to programmer)
- Arguments in favor of (automatic) GC: avoid programming errors (dangling references, memory leaks)

Scope Rules

- Scope: textual region of program in which binding is active
- In most modern languages, scope is statically (compile-time) determined
- In C, a new scope is introduced when:
 - A subroutine starts
 - Entering a new block ({...})
- Runtime creates new bindings for local objects
- Deactivate previous global and/or non-local binding (w.r.t. to current scope)
- On subroutine exit, destroy local bindings, reactivate global/non-local bindings previously active

Scope Rules

- Binding can actually be determined statically (but executed at runtime), i.e., we can look at the code of a program and know which binding will be active
- Other languages (APL, Snobol, Tcl and Lisp) were dynamically scoped:
 - Bindings depend on the flow of execution
- Informally, “scope” also refers to specific program region in which something does not change or where something is in effect
- Examples of scope: a block, a module, a class/object, a structured control-flow construct
- Some languages call the binding process elaboration
 - May include allocation of objects in stack and assignment of initial values

Scope Rules

- Referencing environment:
 - set of active bindings at some point of the program's execution
 - Sequence of scopes that can be examined (in order)
- Binding rules:
 - When a reference to a subroutine is chosen
 - Happens when a reference to a subroutine is stored in a variable, passed as a parameter or returned as a function value (e.g. a lambda function)
 - Deep binding: happens when a subroutine is created
 - Shallow binding: occurs when reference to subroutine is used

```
function f1() {  
    var x = 10;  
    function f2(fx) {  
        var x;  
        x = 6;  
        fx();  
    }  
    function f3() {  
        print x;  
    }  
    f2(f3);  
}
```

Pseudocode Example

Static Scoping

- With STATIC (LEXICAL) SCOPE RULES, a scope is defined in terms of the physical (lexical) structure of the program
- Basic scheme: Current binding for names found in closest surrounding block (e.g. braces, indentation in python)
 - The determination of scopes can be made by the compiler
 - All bindings for identifiers can be resolved by examining the program
 - Typically, we choose the most recent, active binding made at compile time
 - Most compiled languages, C and Pascal included, employ static scope rules
- Several variants of basic scheme

Static Scoping

- Pre Fortran90:
 - Distinguish between global and local variables
 - Scope of local variables limited to subroutine in which they appear
 - Non-declared variables (yes, non-declared) assumed to be local, and of type integer if the name start with letters I-N, real otherwise
- Conventions for successors changed
- Implicit declarations could be turned off from Fortran90 onwards
- Lifetime of local variable normally limited to single execution of subroutine
- Fortran: programmer can explicitly save the value of a variable; similar mechanism to C static variables or Algol own
 - Lifetime of variable expands program execution
 - Name-to-variable binding inactive when subroutine not in execution

Nested Subroutines

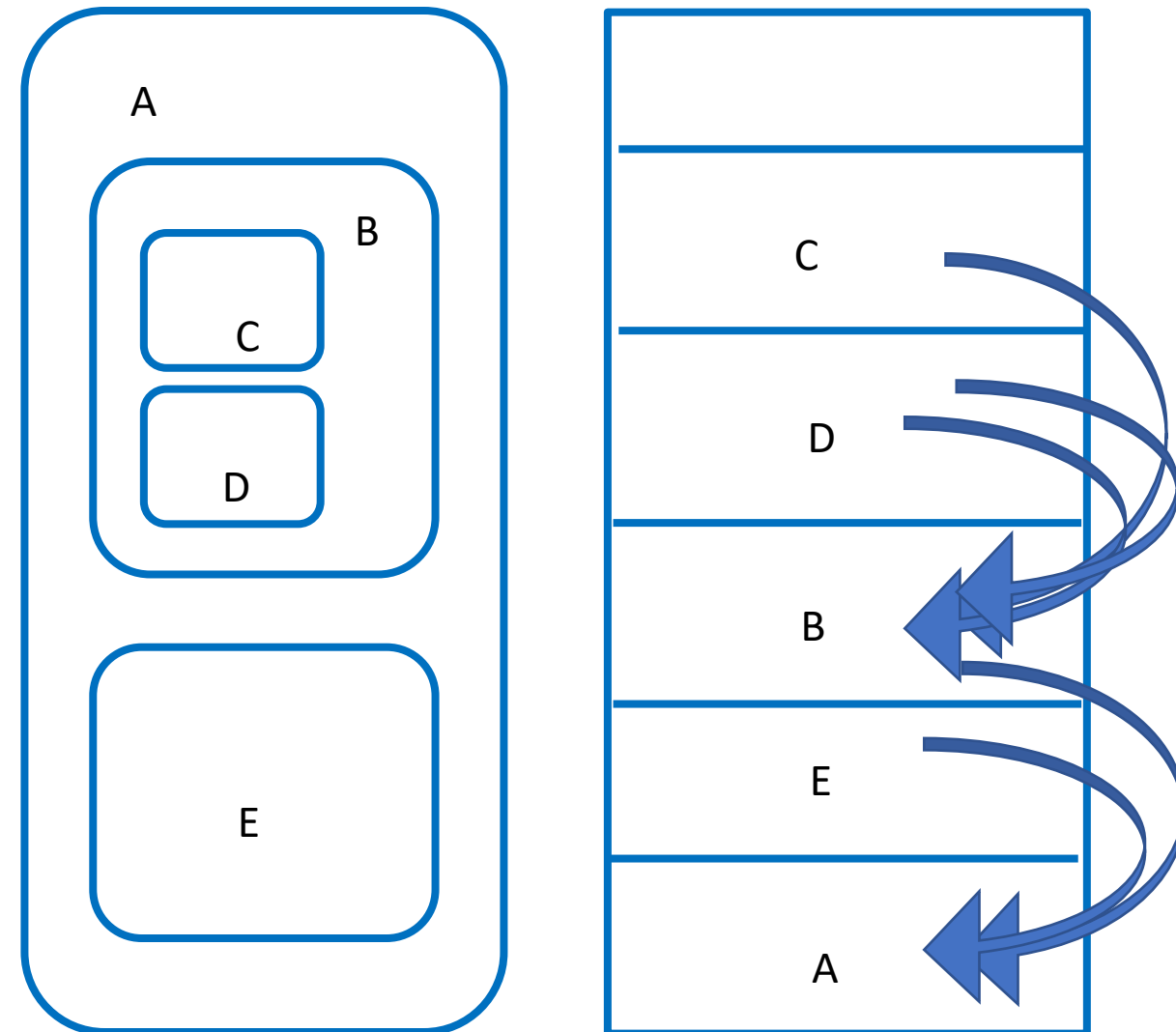
- Ability to define subroutines inside each other
- Introduced in Algol 60
- Feature in several programming languages: Ada, Common Lisp, Python
- Other languages (e.g. C and descendants), allow classes or other scopes to nest
- Algol used "closest nested rule" for bindings
- When a name is hidden by a nested declaration, two options:
 - Hidden name is completely inaccessible
 - If language permits, add a qualifier such as the routine name

```
function
  E(x: real): real;
  function F(y: real):
    real;
    begin
      F := x + y
    end;
begin
  E := F(3) + F(4)
end;
```

Pascal Example

Non-local Access

- Frame pointer (fp) gives access to calling context / activation
- How to access variables in other active frames?
- Problem: frames in activation stack can appear in different order from the nested declaration
- Static links: provide access to outer surrounding subroutine
- Compiler generates code to chase static links at runtime



Declaration Order

- Design question: can an expression E refer to any name declared in the current scope, or only to names declared before E in the scope?
- Some languages (e.g. Algol 60, Lisp) required all declarations to be made at the beginning of the scope (the begin/end or {})
- Pascal case:
 - Changed to allow declarations in the middle of blocks
 - However, scope of declaration remained the whole block
 - Produced weird semantic errors (see book, Sec. 3.3.3)
 - Then modified scope rules to affect only from declaration point onwards (like C)
- Special mechanisms to support recursive types and subroutines: forward declaration:

```
class MyClass;  
  
typedef MyClass myclass_t;
```
- Other languages do not require declarations (e.g. python), variables created on first use

Declarations and Definitions

- Names need to be known and available before being used
- Problem arises in recursive types and subroutines
- What if two structures need to reference each other?
- C and C++ distinguish between declaration and definition
- Declaration: sets name and scope, may skip some details
- Definition: completes object (in the broad sense) description

```
struct node;  
struct tree {  
    struct tree * parent;  
    struct node * first;  
};  
struct node {  
    struct tree * sibling;  
};
```

```
int sum1 (int x);  
int sum2 (int y) {  
    ...  
    return sum1(y);  
}  
int sum1 (int z) {  
    ...  
    return sum2(z);  
}
```

Dynamic Scoping

- Binding depends on control-flow and order of subroutine invocation
- Rule of thumb: current/applicable binding is the one most recently used (and not yet destroyed)
- Side-effect
 - Semantic rules of language dynamically enforced
 - Type-checking and arguments checking deferred to runtime
- Languages with dynamic scope tend to be interpreted

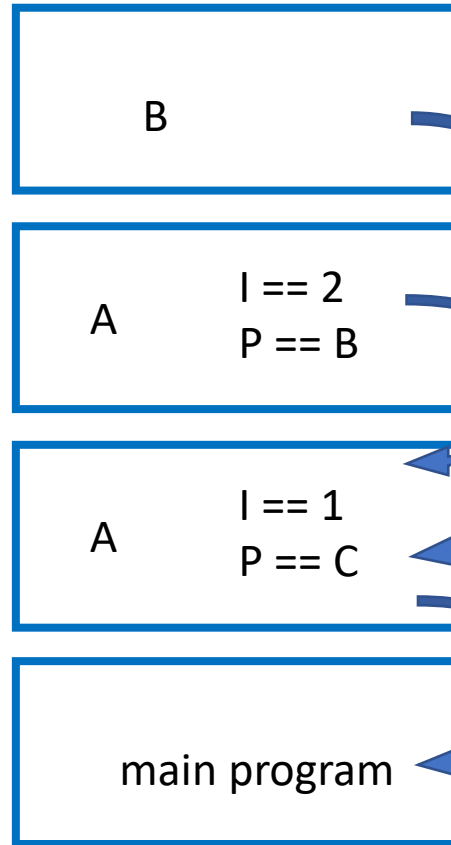
Static vs Dynamic Scope Rules

```
program scopes (input, output );  
var a : integer;  
procedure first;  
begin a := 1;  
end;  
procedure second;  
var a : integer;  
begin  
    first;  
end;  
begin  
    a := 2; second; write(a);  
end.
```

- If static scope rules are in effect (as would be the case in Pascal), the program prints a 1
- If dynamic scope rules are in effect, the program prints a 2
- Why the difference? At issue is whether the assignment to the variable a in **procedure** first changes the variable a declared in the main program or the variable a declared in **procedure** second

Static vs Dynamic Scope Rules

```
def A(I,P):  
    def B():  
        print(I)  
    if (I>1):  
        P()  
    else:  
        A(2,B)  
  
def C():  
    pass #do nothing  
  
A(1,C) # main program
```



- Python example of static scoping and deep binding
- Program prints 1
- If shallow binding were in effect, it would print 2
- Combination of shallow binding and static scoping

Access to Variables with Dynamic Scope

(1) a stack (*association list*) of all active variables

- variable sought from top of stack
- equivalent to searching the activation records on the dynamic chain

(2) central table with one slot for every variable name

- the table layout can be fixed at compile time if variables names cannot be created dynamically
- Otherwise, you'll need a hash function or something to do lookup
- Every subroutine changes the table entries for its locals at entry and exit.

- (1) gives slow access but fast calls; (2) yields slow calls but fast access
- Variable lookup in a dynamically-scoped language corresponds to symbol table lookup in a statically-scoped language
- Because static scope rules tend to be more complicated, however, the data structure and lookup algorithm also have to be more complicated

Binding of Referencing Environments

- Referencing environment: collection of scopes that are examined (in order) to find a binding
- SCOPE RULES: determine the scope collection and its order
- BINDING RULES determine which instance of a scope should be used to resolve references when calling a procedure that was passed as a parameter
 - they govern the binding of referencing environments to formal procedures
- Binding time in languages with static scoping and nesting declaration matters because recursive subroutines might have more than one instance
- Closure: captures the current instance of every object at the time the closure is created

Binding of Referencing Environments

	Deep Binding	Shallow Binding
Static Scoping	Matters when accessing objects that are non-local nor global (i.e. some intermediate nesting)	Doesn't make too much sense
Dynamic Scoping	?	?

Names within Scope

- Names are not necessarily unique
- Name reuse can take several forms: aliases, overloading, polymorphism
- Aliasing:
 - Two or more names that refer to the same object in a program, at the same moment
 - Arises naturally in pointer-based programming languages
 - Pointer aliasing can even disallow later program optimizations
 - C provides language constructs (i.e. restrict) to state that some variables do not alias

```
void stuff (int * a, int * b)
{
    *a += 2;
    *b *= 2;
}
void caller ()
{
    int a = 1;
    stuff (&a, &a);
}
```

```
void stuff (int & a, int * b)
{
    a += 2;
    *b *= 2;
}
void caller ()
{
    int a = 1;
    stuff (a, &a);
}
```

Overloading

Overloading:

- `+`: boils down to different machine instructions, specific to datatype
- In a compiler, overloading can be resolved by returning a list of matching candidates; then choose based on semantic checks (type, number of arguments, etc)
- In Ada, identifiers **oct** and **dec** can refer to predefined months of a enumerated type or to numeric bases; decide with context
 - Can add a disambiguating qualifier such as: `print (month' (oct));`
- In C#, every use of an enumeration constant must be prefixed with the type name: `pb = print_base.oct`

```
void func (int a, int b);  
void func (float a, float b);  
void func (double * arr);
```

Function overloading in C++

Overloading

- Several languages provide mechanisms to:
 - Change the default behavior of operators
 - Define new operators

- In Haskell:

let a @@ b = a * 2 + b

Defines a 2-argument, infix operator named @@

Could also have defined it as:

let (@@) a b = a * 2 + b

In C++:

```
class vector {  
    int * data;  
    int n;  
    ...  
    vector operator+(vector v1, vector v2) { ... }  
    vector * (vector src, int s) { ... }  
};  
...  
vector w, v, z;  
...  
w = v + z;
```

Overloading

- Two other related concepts to overloading: coercion and polymorphism
- Coercion: automatically converts value of one type into another; type casting is an example
- Polymorphism: allow several implementations of subroutines with the same name to behave differently
 - In C++, same function signature (return type and arguments) behaving differently in a hierarchy class

Overloading

```
void func (int a, int b);  
void func (float a, float b);  
void func (double * arr);
```

```
int func (int a, int b) {  
    return a + b + 1;  
}  
float func (float a, float b) {  
    return a + b + 1.5;  
}  
int main () {  
    int x, d = 1;  
    float y, z, a,b,c;  
    a = 1; b = 2; c = 3;  
    x = func (a,b);  
    y = func (a,b);  
    z = func (d,b);  
    printf ("x = %d\n", x);  
    printf ("y = %f\n", y);  
    printf ("z = %f\n", z);  
    return 0;  
}
```

```
class employee {  
    ...  
    float salary () { return 100};  
};  
class manager : public employee {  
    ...  
    float salary () {  
        return employee::salary () * 2;  
    }  
};  
class owner : public manager {  
    ...  
    float salary () {  
        return manager::salary () * 3;  
    }  
};
```