Binding Time

- A binding is an association between a name and the entity it denotes.
- Binding time is the time at which a binding is made.
- In general, early binding leads to greater efficiency, later binding leads to greater flexibility.

Binding Times

Language Design Time: Constructs for control-flow (`for`, `switch`), data aggregating constructors (`array`, `structures`) are chosen at language design time.

Language Implementation Time: Typically the precision of numeric types, handling of runtime exceptions, coupling to the OS, runtime organization of memory are chosen when the language is being implemented.

Program Writing Time: Programmers choose names, data structures and algorithms.

Binding Times C>ontinued

Compile Time: The mapping of high-level constructs to machine code, the memory layout of static data and the memory layout of stack frames.

Link Time: Usually programs consist of many separately compiled units (*object files*), where names in one compilation unit refer to entities defined in other compilation units. These inter-unit references are resolved at link time.

Binding Times Continued

Load Time: The physical (old OS) or virtual (modern OS) address of entities is determined at load time.

Run Time: Bindings of values to variables occur at run time. Run time includes *program startup time*, *module entry time*, *elaboration time* when declarations are first encountered, *subroutine call time*, *block entry time* and *statement execution time*.

Static versus Dynamic

- The word *static* is used to describe to bindings made **before** run time, whereas the word *dynamic* is used to describe bindings made **during** run time.
- Compilers are more efficient than interpreters because they make earlier decisions and predict the address of variables at compile-time.
- Just-In-Time (JIT) compilers can be more efficient than static compilers as they can take advantage of runtime information.
- Certain language features require dynamic capabilities: Smalltalk's *polymorphic* dispatch of a *message* to a *receiver* requires searching a run-time inheritance chain for a super-class of the receiver which understands the message; else *message not understood* error.

Entity and Binding LifeCycle Events

- Creation of entities.
- Creation of bindings.
- References to names representing variables, functions etc. which use bindings.
- Deactivation and reactivation of bindings. For example, a binding lifetime can have a hole when a block which *hides* a variable is active.
- Destruction of bindings.
- Destruction of entities.

Entity and Binding LifeCycle Events Continued

- Binding and entity lifetime not necessarily coincident.
- A binding between a reference parameter to a subroutine and an entity is destroyed when the subroutine terminates, but the entity survives. For example, in C++ `void swap(int &x, int &y)`, the `int`'s bound to by `x` and `y` continue to exist even after a call to 'swap() exits and the binding is destroyed.
- Binding can have longer lifetime than entity as when an entity is `malloc()`'d in C and pointed to by some pointer `p`, but entity is `free()`'d before `p` goes out of scope. For example:

```
{ char *p = malloc(1000);
  ...
  free(p);
  //name p continues to exist
}
```

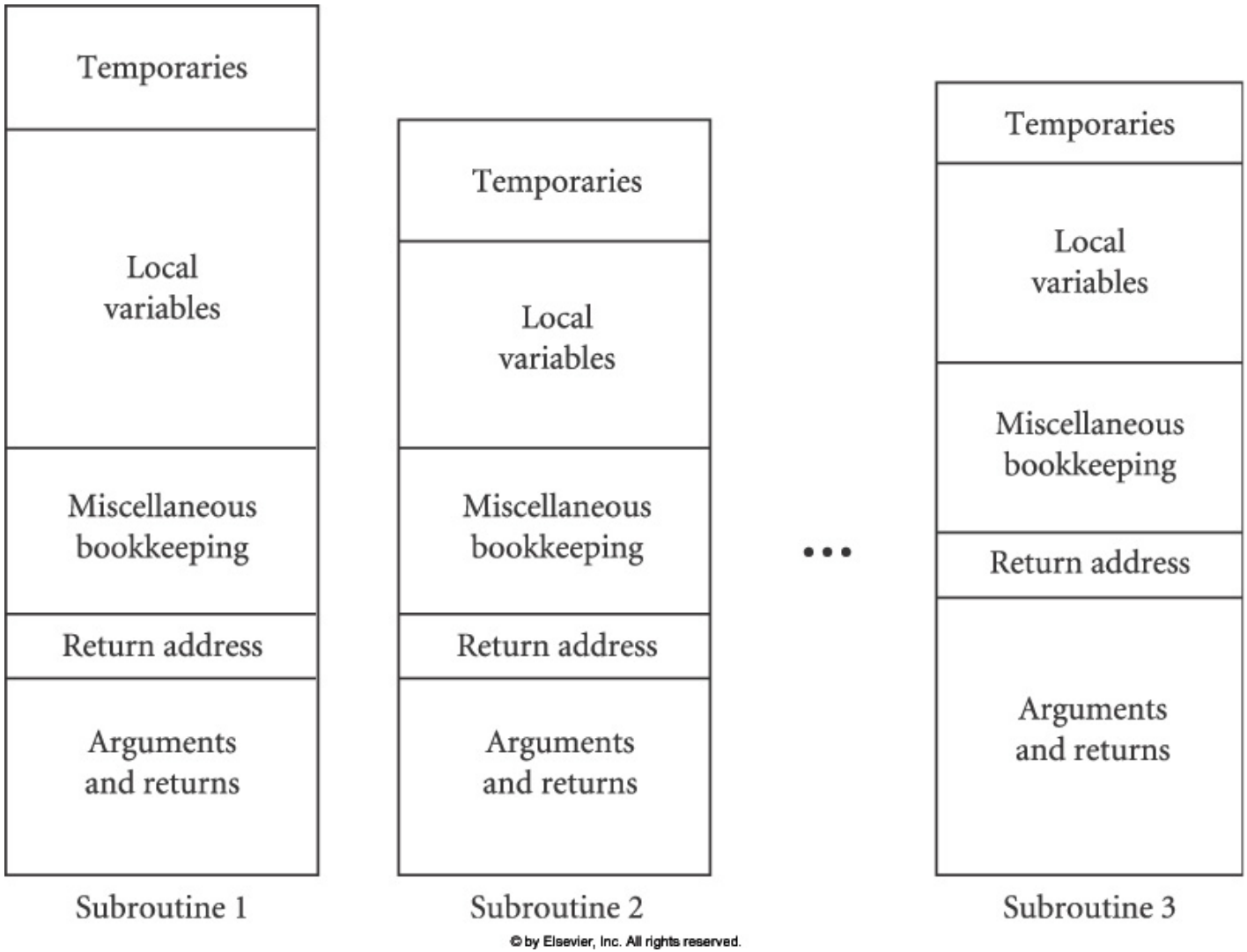
Entity Lifetimes

- Coincident with *lifetime of program*. Can use *static allocation*.
- Coincident with *lifetime of function, subroutine or block*. Can use a LIFO *stack allocation*.
- Indeterminate lifetime. Can use *heap allocation*.

Static Allocation

- Global variables. Normally a bad idea for non-constant data in modern *multi-threaded* programs.
- Constants like numeric and string literals. “Small” constants fit within instructions; “large” constants need separate locations.
- Tables produced by compiler for exception handling, control (case-statements), dynamic type checking, etc.

Static Allocation for Subroutines



Static Space Allocation for Subroutines

Static Allocation for Subroutines Continued

- Primary allocation mechanism in Fortran, pre Fortran-90.
- Recursion impossible.
- Does not make efficient use of memory as some subroutines may never be active simultaneously. Resolved by the programmer in Fortran using `COMMON` and `EQUIVALENCE` statements.

Static/Own Variables in Subroutines

```
unsigned int rand() {
  enum { a = 1664525, c = 1013904223 };
  static unsigned int seed = 1234;
  return seed = a*seed + c;
}
```

Static variables used even within subroutines in languages which allow recursion. Retains value between invocations while not allowing visibility external to subroutine.

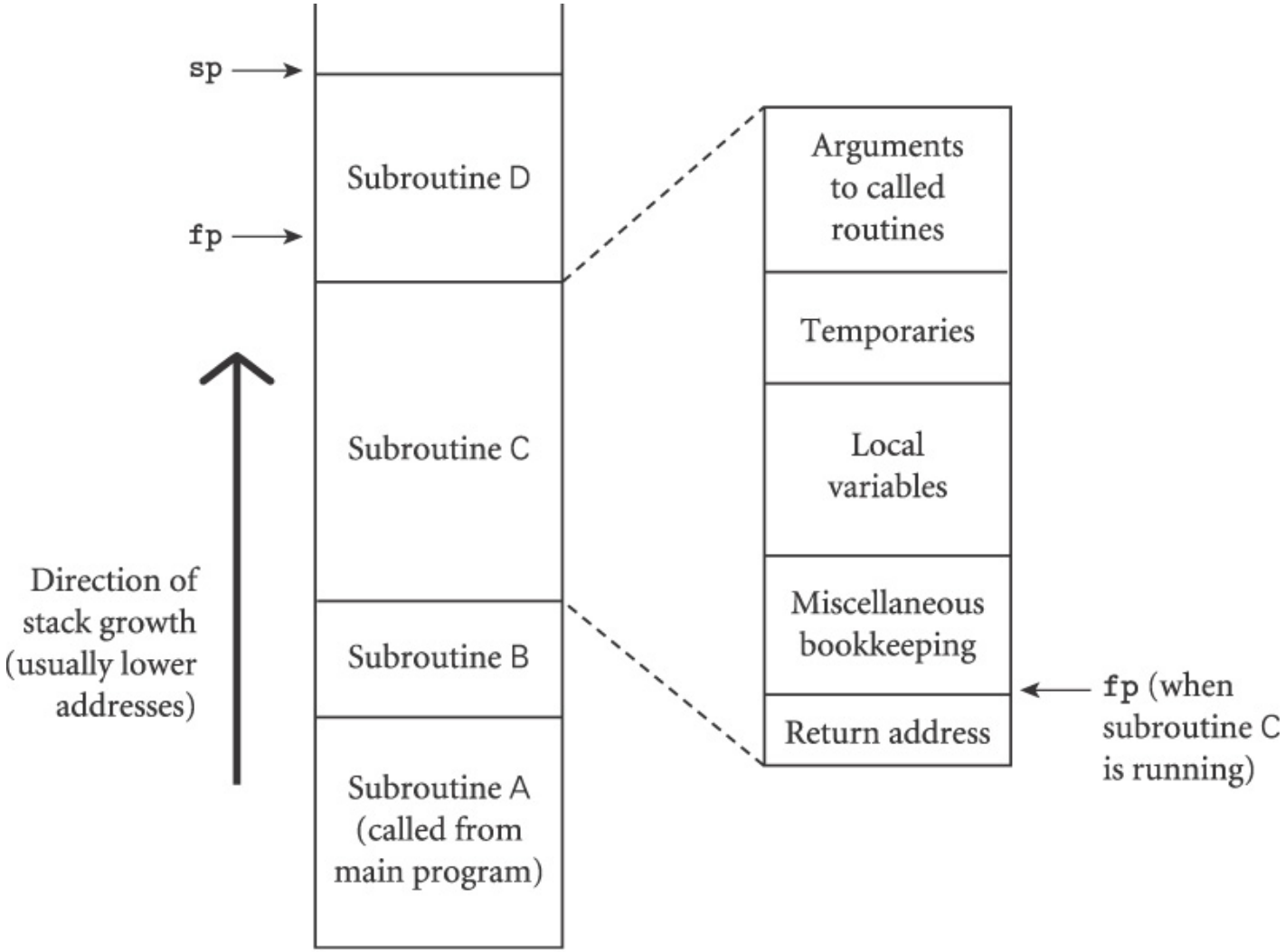
Subroutine Constants

- Need to distinguish between a *compile-time constant* (C# `const`), or constant variables (C# `readonly`) whose values are *final* after *elaboration time*.
- Compile-time constants defined within a subroutine can be allocated statically even if the subroutine is recursive.
- Constant variables cannot be allocated statically if the subroutine is recursive.

Stack Allocation Introduction

- Necessary if a language permits recursion.
- Each instance of a subroutine activation has its own *stack frame* containing arguments (including return value), locals, temporaries and bookkeeping information.
- Details of stack frame layout depend on architecture conventions and compiler.

Stack Allocation Example



Stack Space Allocation for Subroutines

Stack Allocation Continued

- *Calling sequence* consists of code executed by caller immediately before and after the call and the *prologue* code executed by the callee at the start of the subroutine and the *epilogue* code executed by the callee at the end of the subroutine.
- Location of frame on stack cannot be predicted at compile time, but the offset from the frame-pointer `fp` of entities within the frame can be calculated at compile time.

Stack Allocation Continued

- Typically, stack grows from high memory towards low memory. Hence typically, arguments have positive offsets from `fp`, whereas locals and temporaries have negative offsets.
- Even with non-recursive subroutines, stack allocation is preferred because memory is allocated only for active subroutines instead of for all subroutines as would be the case with static allocation.
- With support from all modern architectures, stack allocations is generally a big win.

Heap Allocation

- Heap allocation is used for entities with arbitrary lifetime and dynamically resized entities.
- Space and time considerations in managing a heap.

Fragmentation

- *Fragmentation* occurs because the algorithm uses to allocate heap space makes certain fragments of memory unavailable.
- *Internal fragmentation* occurs when the algorithm allocates a block larger than the request, with the excess portion of the block wasted.
- *External fragmentation* occurs when small free memory blocks are scattered throughout the heap. It may be impossible to satisfy a allocation request even though the requested size is less than the total amount of free memory.

External Fragmentation

External Fragmentation

Heap Allocation Strategies

- Manage free-list of free memory blocks . . . initially a single block containing the entire heap.
- *First-fit* algorithm allocates the first block which is large enough.
- *Best-fit* allocates block which fits exactly or with smallest leftover.
- *Next-fit* starts scan from where previous allocation request succeeded.

Heap Allocation Strategies

- When allocated block is larger than requested size, need to decide between internal and external fragmentation.
- Free list can be unordered, ordered by address (facilitating *coalescing*), ordered by increasing size (facilitating *best fit*), ordered by decreasing size (facilitating *worst fit*).
- Buddy allocation uses multiple free-lists for blocks of specific sizes (powers of 2 or Fibonacci sequence) and rounds up allocation requests to available sizes. If free list is empty, split block from next higher free list.

Programmer Managed versus Automated Memory Management

- Languages like C, C++ require programmer to explicitly deallocate an entity when it is no longer in use. Leads to *dangling pointers* and other hard to debug memory bugs.
- Most functional languages and more recent imperative languages like Java and C# use garbage collection where an entity is free'd automatically by the run-time when the run-time can prove that there cannot be any live references to the entity.

Garbage Collection

- GC used to be inefficient (*stop-the-world*!!) and was relegated to languages like Lisp, but GC technology has advanced so that now it is available in mainstream languages.
- It is worth emphasizing that GC'd languages can still *leak* memory if used carelessly. It is important to ensure that references to unused data-structures be `null`'d out.

Scope

- A *scope* is a program region of maximal size where no bindings change (or at least, none are destroyed).
- Languages like Ada use the term *elaboration* to refer to the process of activating declarations when a scope is entered. In Ada, elaboration can include error-checking, heap-space allocation, exception propagation, creation of concurrent tasks.
- *Static* (*lexical*) scoping versus *dynamic* scoping.

Fortran 77

- No recursion.
- Distinguishes between local and global variables.
- Unless explicitly `save`'d, local variables have a lifetime equal to a single subroutine call (though the common static allocation implementation results in a lifetime equal to that of the program).
- Implicit declaration of undeclared variables; if name starts with `[I-N]`, then `INTEGER` else `REAL`.

Managing Memory in Fortran

- Global variables can be partitioned into `COMMON` blocks which can then be imported by subroutines. Helps separate compilation.
- Each subroutine redeclares the contents of each common block; no check to ensure redeclaration consistency. Hence possible to have two different variables share the same location within a `COMMON` block.
- Similar effect can be achieved by using the `EQUIVALENCE` statement.

Nested Scopes

- *Closest nested scope rule* for resolving bindings: a name is known in the scope in which it is declared and in each internally nested scope unless *hidden* by the redeclaration of the same name in an intervening scope.
- Ada, Common Lisp, ML, Pascal, Scheme allow nested subroutines.
- C does not allow nested subroutines, but allows nested blocks.

Nested Scopes

- Some languages like Pascal require local declarations at the start of a subroutine, others like C89 at the start of a block, others like C++ and Java at any point in a block.
- Usually, scope of declaration is from point of declaration until end of block, but Javascript *hoists* declarations to start of function.
- Predefined constants and functions are defined in a implicit outermost scope. This allows the programmer to override these names with alternate declarations in inner scopes.

Nested Subroutines in Pascal

```
procedure P1(A1: T1);
var X: real;
  ...
  procedure P2(A2: T2);
    ...
    procedure P3(A3: T3);
      ...
      begin (* body of P3 *) end;
    begin (* body of P2 *) end;
    ...
```

Nested Subroutines in Pascal

```
  procedure P4(A4: T4);
    ...
    function F1(A5: T5) : T6;
    var X : INTEGER; (* hides X in P1 *)
    ...
    begin (* body of F1 *) end;
    ...
  begin... (* body of P4 *)
  end;
  ...
begin (* body of P1 *) end.
```

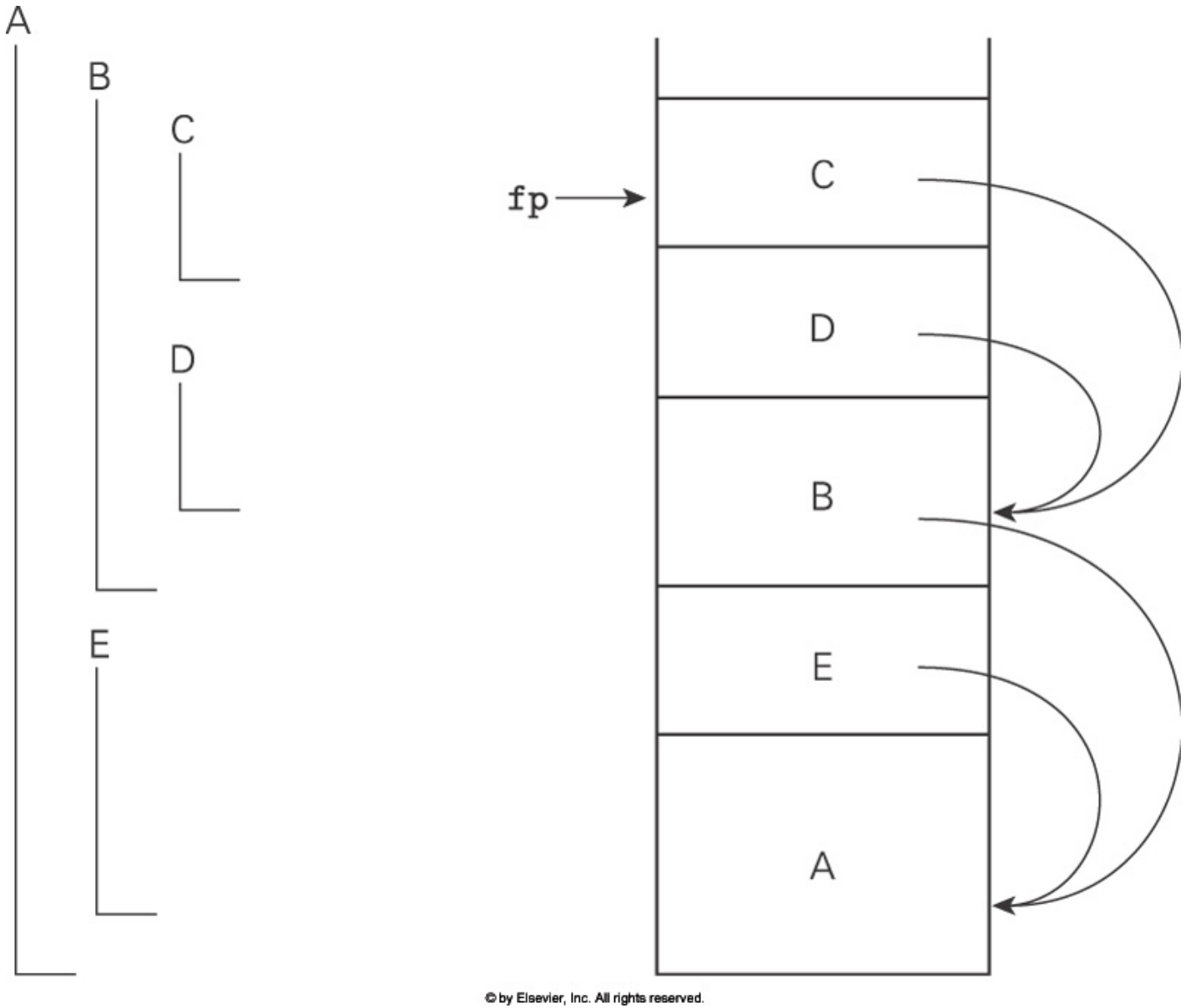
Nested Subroutines

- Inner declaration of a name *hides* or *shadows* a outer declaration of the same name; we say there is a *hole* in the outer scope.
- Some languages like Ada allow syntax like `P1.X` to refer to hidden declarations in an outer scope.
- In C++, `::X` refers to global `X`, irrespective of any local variables names `X`.

Access to Nonlocal Entities

- Local entities accessible via stack frame.
- Since a subroutine is only visible to routines in the same scope or nested scopes, non local entities must be available in an earlier stack frame.
- Keep track of the difference in depth between the current subroutine and non-local subroutine to find stack frame for non-local subroutine.
- In the presence of recursion, we need a *static chain* which links latest stack frames for each depth from inner to outer.

Access to Non-Local Entities



Static Chains

Declaration Order

- Does the scope of a declaration begin at the point of declaration (C, Java) or at the start of the block containing the declaration (Pascal).
- Many languages require *declaration before use* for local variables.
- Pascal requires declaration before use for functions/procedures. Mutual recursive routines need *forward declarations*. Facilitates *single-pass* compilation.
- Java, C++ allow class members to be visible in all methods irrespective of declaration order; classes can be declared in any order and still refer to each other.

A Pascal Anomaly

Pascal defines scope of declaration as procedure containing declaration and it also requires declaration before use. Results in the following anomaly (many Pascal compilers do not catch this error):

```
const N = 10;
...
procedure foo;
const
  M = N; (* semantic error because
            use before declaration *)
  ...
  N = 20; (* declaration which hides outer N *)
```

Declaration Order Immaterial

- In Modula-3, scope of declaration is entire block in which declaration occurs (modulo holes); declaration order does not matter. Hence one declaration can refer to a variable in a subsequent declaration.
- Scripting language Python does not have any declarations. If there is assignment to some local variable, then that is taken as a declaration of S. However a reference to the variable before the assignment will trigger a `NameError`.

Javascript Variable Hoisting

- Variables declared using `let` act like traditional block-scoped variables.
- Variables declared using `var` are hoisted to start of function.
- Variable hoisting can cause surprising behavior.
- Traditionally Javascript had only `var` declarations; `let` was added later.

Javascript Variable Hoisting Example

In let-var.js

```
19  function use_var(n) { //
20    var ret = new Array();
21    for (var i = 0; i < n; i++) {
22      ret.push(function() { console.log("i = " + i);
23    }
24    return ret;
25  }

32  function use_let(n) { //
33    let ret = new Array();
34    for (let i = 0; i < n; i++) {
35      ret.push(function() { console.log("i = " + i);
36    }
37    return ret;
38  }
```

Javascript Variable Hoisting Example Continued

```
7  function run(msg, fns) { //
8    console.log(msg)
9    for (let i = 0; i < fns.length; i++) {
10     fns[i].call();
11   }
12 }

42 const N = 5; //
43
44 run("using var", use_var(N));
45
46 run("using let", use_let(N));
```

Javascript Variable Hoisting Example Log

```
$ ./programs/let-var.js
using var
i = 5
i = 5
i = 5
i = 5
i = 5
using let
i = 0
i = 1
i = 2
i = 3
i = 4
$
```

Declarations versus Definitions

- In C, a entity can be declared multiple times (consistently), but can be defined only once.
- A function with a body constitutes a definition of the function.
- A global variable constitutes a definition depending on the initialization model. Safest to give it a initializer to make it a definition.

Declaration versus Definition in C

In header file:

```
struct Stack; /* declaration */

void pushStack(struct Stack *stack, int value);
int popStack(struct Stack *stack);
```

Declaration versus Definition in C Continued

In implementation file:

```
enum { MAX_STACK = ... };
struct Stack { /* definition */
  int stackIndex;
  int contents[MAX_STACK];
}
void pushStack(struct Stack *stack, int value) {
  ...
}
int popStack(struct Stack *stack) { ... }
```

Declaration Statements

- Pascal allows local declarations only at start of a subroutine.
- Algol 60, C89 and Ada allow local declarations only at the start of a block.
- Algol 68, C9x. Java, C# allow local declarations anywhere where a statement may appear (Java, C# make it illegal to hide outer local declarations). This facilitates locality and initialization during declaration.

Information Hiding

- Minimize visibility of entities to portions of program which do not need them.
- Reduces *congitive load* on the programmer by reducing the amount of information required to understand any portion of the program.
- The use of a local `static` declaration as in the `rand()` example ensures that `seed` is not visible outside `rand`.
- Information hiding provided by block-scope is limited.

Modules

- A *module* is collection of entities (subroutines, variables, types, etc.) which are encapsulated such that entities within the module are visible to each other but are not visible outside the module unless explicitly *exported*. Entities external to the module may not be visible unless explicitly *exported*.
- Modules can be found in Modula 1, 2, 3, CLU (*clusters*), Turing, Ada (*packages*), C++ (*namespaces*), Java (*packages*).

Separate Compilation

- Modules consist of a declaration part or header and an implementation part or body.
- Module clients need only the header and are independent of the body.
- Because of consequent information hiding, the body can be reimplemented without changing module clients.
- Usually module type is exported in header as a *pointer* or *reference* to an *opaque* structure. Since size of a pointer/reference is fixed, clients can be compiled without needing to know details of module type.

Modules in C

C has no explicit module support. However, it is possible to implement modules by following some conventions:

- For each module *module*, have a specification *module*`.h` file which simply declares the types and functions exported by the module.
- For each module *module*, have a implementation *module*`.c` file which implements the types and functions exported by the module. Private functions/data for the module should be declared using the `static` storage-specifier which ensures that the declarations are local to the implementation file.
- Types whose details are not required externally can be declared using a opaque `struct` declaration. This is possible if module function parameters or return values only use pointers to the type.

C Specification of Single Instance Stack

```
typedef void *Element; //generic type

//push element onto single stack instance
void push(Element element);

//return element popped from single stack instance
Element pop();
```

C Implementation of Single Instance Stack

```
enum { MAX_SIZE = ... };

//static var's not visible outside file
static Element stack[MAX_SIZE];
static int top = 0;

void push(Element element) { ... }
Element pop() { ... }
```

C Specification of Multiple Instance Stack

```
typedef void *Element; /* generic type */
struct Stack; //opaque type

//constructor: allocate and return initialized stac
struct Stack *newStack();

//destructor
void freeStack(struct Stack *stack);

//push element onto stack
void push(struct Stack *stack, Element element);

//return element popped from stack
Element pop(struct Stack *stack);
```

C Implementation of Multiple Instance Stack

```
enum { MAX_SIZE = ... };
struct Stack { //define opaque type
  int top;
  Element stack[MAX_SIZE];
}
```

C Implementation of Multiple Instance Stack Continued

```
struct Stack *newStack() {
  struct Stack *stack =
    malloc(sizeof(struct Stack));
  stack->top = 0;
  return stack;
}

void freeStack(struct Stack *stack) {
  free(stack);
}
```

C Implementation of Multiple Instance Stack Continued

```
void push(struct Stack *stack, Element element) {
  if (stack->top < MAX_SIZE) {
    stack->stack[stack->top++] = element;
  }
  else error();
}

Element pop(struct Stack *stack) {
  if (stack->top > 0) {
    return stack->stack[--stack->top];
  }
  else error();
}
```

Open versus Closed Scopes

- Scopes into which all names from external scopes are automatically imported are *open scopes*.
- Scopes into which names from external scopes must be explicitly imported are *closed scopes*. Hence modules are closed scopes.
- Nested subroutines and blocks are open scopes in most Algol family languages.
- Languages like Java, C#, Ada have *selectively open scopes*. Exported symbol `foo` in module/package `A` can be accessed in client as `A.foo`, or simply as `foo` if explicitly imported.

Dynamic Scope

- With *dynamic scope*, the current binding for a given name is the one encounted most recently during *execution* and not yet destroyed by returning from its scope.
- Depends on the flow of control at run-time and on the order in which subroutines are called.
- Languages with dynamic scope include classical Lisp, APL, Perl (before Perl 5 which allows static scope), Snobol.
- Scheme, Common Lisp are statically scoped, though the latter allows optional dynamic scope via *special variables*.

Dynamic vs Static Scope: Perl Example

```
3  #var declared local is a dynamically-scoped var.
4  local $dynamic = 2;
5
6  #var declared using my is a lexically-scoped var.
7  my $static = 2;
8
9  sub out($$$) {
10   my($msg, $stat, $dyn) = @_;
11   print "$msg: static=$stat, dynamic=$dyn\n";
12 }
13 sub f() {
14   local $dynamic = 3;
15   my $static = 3;
16   h();
17 }
```

Dynamic vs Static Scope: Perl Example Continued

```
19  sub h() {
20    my $caller = (caller(1))[3];
21    out("in h() called from $caller",
22        $static, $dynamic);
23  }
24
25  sub go() {
26    out("before f()", $static, $dynamic);
27    f();
28    out("after f()", $static, $dynamic);
29    h();
30  }
31
32  go();
```

Dynamic vs Static Scope: Perl Example Log

```
$ ./programs/static-dynamic.pl
before f(): static=2, dynamic=2
in h() called from main::f: static=2, dynamic=3
after f(): static=2, dynamic=2
in h() called from main::go: static=2, dynamic=2
$
```

- When `h()` called from `f()`, it accesses value of `$dynamic` defined in `f()`.
- Once `f()` returns, `$dynamic` automatically restored to its previous value.

Dynamic Scope Tradeoffs

- Allows customization of routines at run time. For example, `printInteger()` with base set by run time variable `printBase`. Default `printBase` to `10` with callers who want different base setting it appropriately.
- Difficult to understand program without understanding dynamic flow.
- Use a static `printBase` with callers who want non-default value setting and restoring value. Dynamic scope provides automatic restore.

Scope Implementation

- Have a symbol table which maintains mapping between names and information about the entity bound to that name.
- Very often, information is never deleted from symbol table (LeBlanc and Cook approach).
- Operations on symbol table include `lookup(String name)`, `add(String name, SymInfo info)`, `beginScope()` and `endScope()`.
- Give each scope a number and use a *scope stack* to denote current referencing environment.

Association Lists for Dynamic Scope

- Association list or *alist* is a list of pairs, giving bindings for identifiers. The current value of a identifier is the first value found when searching down the list.
- When execution enters/leaves a scope, bindings for the names defined within that scope are pushed/popped onto the alist.
- Scope entry/exit relatively efficient; lookup can be inefficient.
- Called *deep binding* by Lisp implementers because lookup can be deep.

Table for Dynamic Scope

- Have a table with a slot for each name with each slot containing a stack of bindings for that name.
- Name lookup is easy: at head of stack associated with the slot for the name.
- Entry/exit of scope is more complex as it involves pushing/popping entries at the slots for all names defined in that scope.
- Called *shallow binding* by Lisp implementers because lookup is shallow.

Deep versus Shallow Binding

Under dynamic scope:

- When a function is passed as a parameter, when is its referencing environment created?
- If the referencing environment is created when the reference is created then we have *deep binding*.
- If the referencing environment is created when the function is actually called then we have *shallow binding*.

Program for Binding Rules

```
typedef struct
    ...
    age : integer
    ...
} Person;
int threshold;
Person[] people;
```

Program for Binding Rules Continued

```
boolean olderThan(const Person &p) {
  return p.age > threshold;
}
void printPerson(const Person &p) {
  /* use lineLength; */
}
```

Program for Binding Rules Continued

```
void
printSelectedPeople(Person[] people,
                    boolean (*pred)(Person),
                    void (*print)(Person&)) {
  int lineLength = (istty(stdin)) ? 80 : 132;
  foreach (Person p : people) {
    if (pred(p)) print(p);
  }
}

printSelectedPeople(people, olderThan,
                    printPerson);
```

Deep versus Shallow Binding

Under dynamic scope:

- For `lineLength` need last binding of `lineLength`: *shallow binding*.
- For `threshold` need value when `olderThan` is defined: *deep binding*.
- The need for deep binding in early dynamically scoped Lisp's was referred to as the *funarg problem*.

Closures

- When a function is returned in a lexically-scoped programming language, returned value must retain a reference to the environment within which returned function is defined.
- A closure is a pair $\langle$code-pointer, env-pointer$\rangle$.
- Since alists are non-destructive, closure can be easily represented using alists, with referencing environment pointing to alist giving environment when closure is created.
- For table approach, need to build environment containing first entry of every slot in table, though optimizations are possible.

Deep Binding in Pascal

```
program binding(input, output);
procedure A(I:integer; procedure P);
  procedure B;
    begin writeln(I); end;
  begin (* A *)
    if I > 1 then P else A(2, B);
  end;
```

Deep Binding in Pascal Continued

```
procedure C; begin end;
begin (* main *)
  A(1, C);
end.
```

Deep Binding in Pascal Continued

- The value of `I` used is that in effect when closure for `P` is created in initial invocation of `A`: 1.
- A closure in statically-scoped language captures current instance when closure is created.
- When closure's routine called, it uses captured instances, even though there may be newer instances created by recursive calls.

Class of Values

First-class Values: Can be passed as a subroutine parameter, returned from a subroutine, stored in a variable, and optionally, be created at run time. Eg: integers and characters in most programming languages.

Second-class Values: Can be passed as a subroutine parameter, but cannot be returned from a subroutine or assigned to a variable. Eg: subroutines in most programming languages.

Class of Values

Third-class Values: Cannot be passed as subroutine parameter or returned by subroutine or assigned to a variable. Eg: labels in most programming languages.

Most functional languages have subroutines as first-class values.

First-Class Functions with Static Scope

- If a function is returned from another function, then we need to preserve the referencing environment.
- Hence a language with first-class functions needs to have locals with *unlimited extent*, conceptually allocated on the heap.
- Most imperative languages use stack-based allocation for local. Avoid above problems by language restrictions: no nested routines for C, C++, Fortran, Modula-2 allows references only to outermost subroutines, Modula-3 only allows outermost subroutines as return values.

Balance Program

In Ruby program balance.rb:

```
3  def new_account(balance)   #
4    #returns hash of functions
5    { withdraw:
6        lambda { |amount| balance -= amount },
7      deposit:
8        lambda { |amount| balance += amount },
9    }
10 end
11
12 #create 2 new acounts
13 a1, a2 = new_account(100), new_account(150)
14 print a1[:withdraw].call(24), "\n"
15 print a2[:deposit].call(20), "\n"
16 print a1[:withdraw].call(26), "\n"
```

Balance Program Log

```
$ ./programs/balance.rb
76
170
50
$
```

Balance Program Retrospecive

- Closures can be used to achieve same kind of encapsulation as typed-modules.
- This style of programming is encouraged in the seminal *Structure and Interpretation of Computer Programs*.

Aliases

- Multiple names (*aliases*) refer to a single entity within a single scope.
- Aliases introduced with `COMMON` and `EQUIVALENCE` statements in Fortran; variant records and `union`'s in languages in Pascal and C.
- Aliasing situations often occur implicitly in the presence of *references* and *pointers*.

C++ Reference Aliases Example

```
double sum, sumOfSquares;
void accumulate(double &x) {
  sum += x;
  sumOfSquares += x*x;
}
accumulate(sum);
```

C Potential Pointer Alias Example

```
int a, b, *p, *q;
...
a = *p;
*q = 3;
b = *p;   /* can we reused *p from before? */
```

Can we reuse previous value of `*p`. Only if `*q` does not alias `*p`.

Alias Analysis in C

- Recent sophisticated *alias analysis* algorithms have made C code run as efficiently as Fortran.
- C99 allows the programmer to use the `restrict` keyword to declare a no-alias declaration.

Overloading

- A single name can reference multiple entities within the same scope.
- Disambiguation is usually via syntactic or semantic context.
- Symbol table module in compiler handles overloading by returning list of entities for a name in a particular scope.

Name Spaces in C

In C, the same name overloading is via namespaces which are:

- Entities (variables), functions, `typedef` and `enum`-constants.
- Labels.
- `struct`, `union` and `enum` tags.
- A separate name space for the fields of each `struct` or `union`.

C Namespaces Example

```
int f(int a) {
  typedef struct f {
    int f;
    struct f *a;
  } F;
  F s;
  f: /* label */
  ...
}
```

Function Overloading

- In Java, C++, C# we can have the same name for different functions provided each use can be disambiguated by the number and types of the arguments.
- In Ada, we can have the same name for different functions provided each use can be disambiguated by the number and types of the arguments and the return type.

C++ Function Overloading

```
struct Complex { ... };
enum Base { bin, oct, dec, hex };
void printNum(int n) ...
void printNum(int n, Base b) ...
void printNum(Complex c) ...
```

C++ Function Overloading

```
int i;
Complex x;
printNum(i);       //1st function
printNum(i, dec); //2nd function
printNum(x);       //3rd function
```

Operator Overloading

- Most programming languages allow binary operator + to refer to both integer addition as well as floating-point addition. Similarly for other arithmetic operators.
- Some languages like C++, Prolog, ML allow programmer to reuse built-in operators or define new operators for other functions. So in C++, it is possible to overload the arithmetic operators to operate on a user-defined `Complex` type.

References

Text, Chapter 3.