Notes for CSC 2/454, Feb. 6 and 11, 2025

==========

Intro to Naming: Scope, lifetime, bindings, and storage management

A ***name*** is exactly what you think it is.
Most names are identifiers, though symbols (like '+') can also be names.

A ***binding*** is an association between two things, such as a name and the thing it names.

The ***scope*** of a binding is the part of the program (textually) in which the binding is active.

Binding time is the point at which a binding is created or, more generally, the point at which any implementation decision is made.
Examples include

- language design time
  - program structure, possible types
- language implementation time
  - I/O, arithmetic overflow, type equality (if unspecified in manual)
- program writing time
  - algorithms, names
- compile time
  - plan for data layout
- link time
  - layout of whole program in memory
- load time
  - choice of physical addresses
- run time
  - value/variable bindings, sizes of strings
  - subsumes
    - program start-up time
    - module entry time
    - elaboration time (point at which a declaration is first "seen")
    - procedure entry time
    - block entry time
    - statement execution time

The terms ***static*** and ***dynamic*** are generally used to refer to things bound before run time and at run time, respectively. Clearly "static" is a coarse term. So is "dynamic."

What gets bound when varies from language to language.

It is difficult to overstate the importance of binding times in programming languages.

In general, early binding times are associated with greater efficiency.
Later binding times are associated with greater flexibility.
Languages with lots of early binding tend to be compiled.
Languages with lots of late binding tend to be interpreted.

Today I want to talk in particular about the binding of identifiers to the things they name. I'll use the name "item," informally, for anything that can have a name.

Scope and Lifetime

Fundamental to all programming languages is the ability to name things, i.e., to refer to items using symbolic identifiers rather than values,addresses, etc.
Items we might name include

- constants
- variables
- functions
- parameters
- modules
- classes
- fields
- types
- exceptions
- labels
- threads
- ...

Anything that isn't figured out until run time (values of variables and parameters in particular) has to be represented by data (bits) in memory. Some but not all data have names.

Dynamic storage in C, Ada 95, or Fortran 90, for example, is referenced through pointers, not names. Similarly, dynamic storage in Java or C# is referred to indirectly through references.

The ***lifetime*** of an item runs from when the space for it is allocated until it is reclaimed. The ***lifetime of a binding*** runs from when the name is first associated with the item until it is no longer associated with it (and never will be again). A binding may not be ***active*** (usable) throughout its lifetime: it may be hidden by a nested use of the same name, or it may be valid only when running a given function or a method of a given class.

Typical timeline:

creation of item
  creation of binding
    uses of name that is bound to item
      (temporary) deactivation (hiding) of binding
      reactivation of binding
    more uses
  destruction of binding
destruction of item

If an item outlives its binding it's ***garbage***.
If a binding outlives its item it's a ***dangling reference***.

The ***scope*** of a binding is the textual region of a program in which the binding is active. In most but not all languages this scope is determined at compile time.

That is, nothing has to happen at run time to activate and deactivate bindings; the compiler has already figured out what's visible where.
In such languages, scope is sometimes called ***lexical extent***; lifetime is sometimes called ***dynamic extent***.

(More later on the rules that determine scope.)

In addition to talking about the "scope of a binding," we sometimes use the word 'scope' as a noun all by itself, without an indirect object. A "scope" is a program region of maximal size in which no bindings are destroyed.

In many, but not all languages, the scope of a binding is determined by a ***declaration***. From the perspective of formal semantics, the declaration can be thought of as code that actively establishes visibility, even if the compiler is smart enough to do all the work ahead of time.

Algol 68 introduced the term ***elaboration*** for the "execution" of declarations. It's a useful concept because some declarations do more than establish bindings, and some of the extra stuff has to happen at run time. Elaboration can

- allocate space
- perform dynamic semantic checks (is the lower bound of this array <= the upper bound?) and perhaps raise an exception
- start a thread
- ...

And in some languages (e.g., Python & Ruby), declarations really ***are*** executed:

```
class foo
     if A > B
          method bar() ...
     else
          method bar() ...
```

In most languages with subroutines, we ***open*** a new scope on subroutine entry. We create bindings for new local variables, deactivate bindings for global variables that will be hidden by local ones (the globals are said to have a "***hole***" in their scope), and then make references to variables. On subroutine exit, we destroy bindings for local variables and reactivate bindings for nonlocal variables that were deactivated.

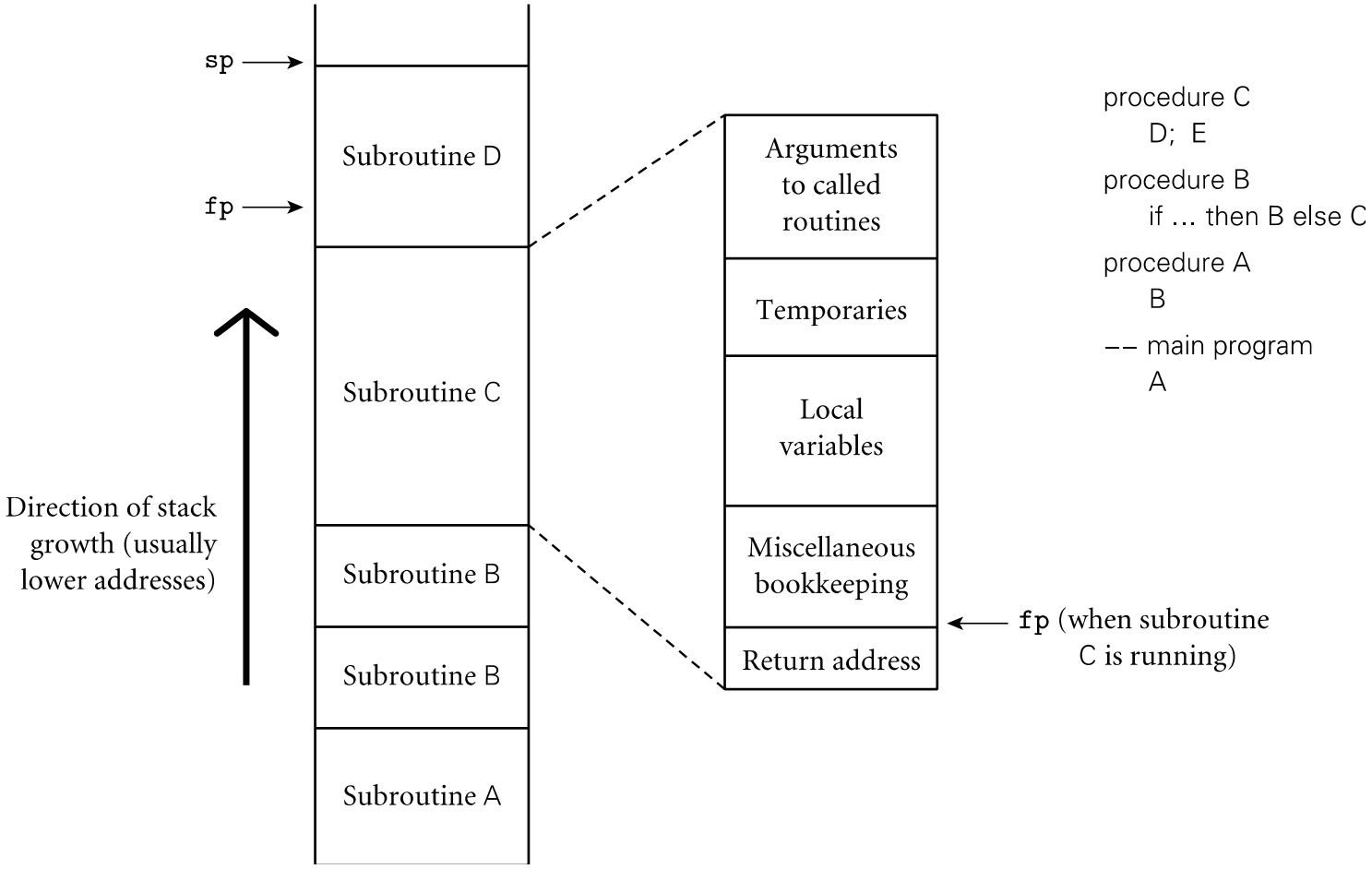
The ***referencing environment*** of a statement or expression is the set of active bindings. A referencing environment corresponds to a collection of scopes that are examined (in order) to find a binding. ***Scope rules*** determine that collection and its order.

Storage Management -- for items with various lifetimes.

- Static allocation for
  - code
  - globals
  - own/static variables
  - explicit constants (strings, sets, other aggregates)
    - some scalars may be global;
    - others may simply be embedded in instructions
- Central stack (chap. 9) for
  - parameters
  - local variables
  - temporaries
  - bookkeeping information
- Why a stack?
  - allocate space for recursive routines
  - reuse space
  - minimize management overhead
- Heap (chap. 7) for
  - dynamic allocation

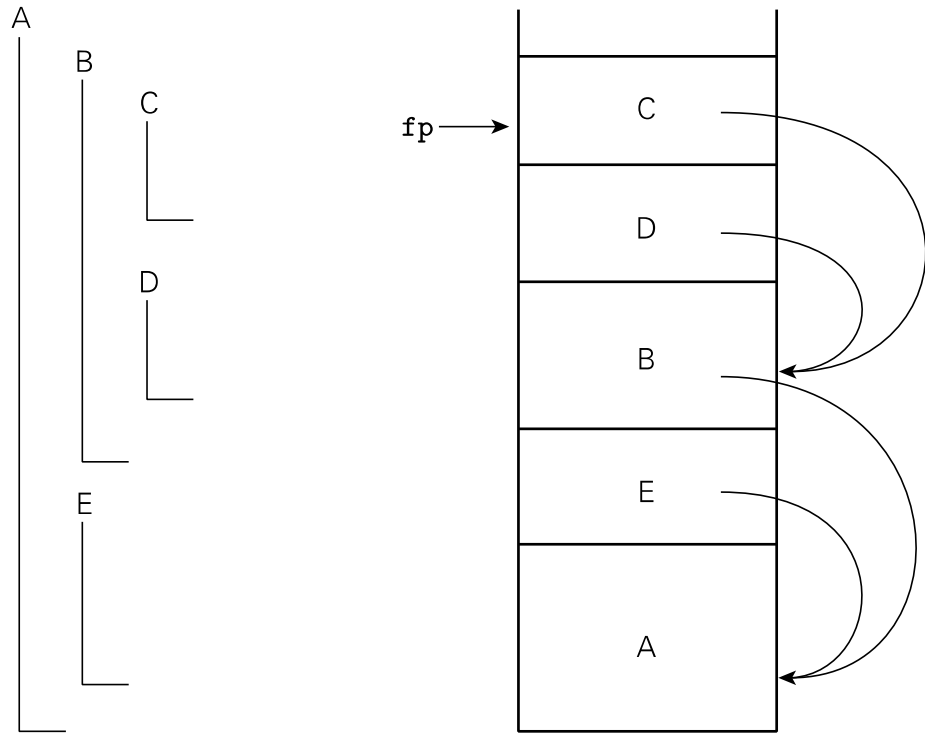
Maintaining the run-time stack

- Contents of a stack frame
  - bookkeeping: return PC (dynamic link), saved registers, line number, static link, etc.
  - arguments and returns
  - local variables
  - temporaries
- Maintenance of stack is responsibility of “calling sequence” and subroutine “prologue” and “epilogue” (more on this in Chap. 9)
  - space is saved by putting as much in the prologue and epilogue as possible
  - time ***may*** be saved by putting stuff in the caller instead, or by combining what’s known in both places (interprocedural optimization)



Local variables and arguments are assigned fixed ***offsets*** from the stack pointer or frame pointer at compile time.

Access to non-local variables is usually implemented using ***static links***. Each frame has a pointer to the frame of the (correct instance of) the routine inside which it was declared. In the absence of formal subroutines, "correct" means closest to the top of the stack. You access a variable in a scope k levels out by following k static links and then using the known offset within the frame thus found.



NB: many languages allow you to declare nested scopes ***within*** the body of a subroutine. (OCaml, for example, does this all the time.) Declarations in these nested scopes hide outer variables with the same name, just as declarations at the tops of subroutines do. These nested scopes are generally considered to be a good idea, esp. since the implementation can roll space management into that of the surrounding routine: then the run-time overhead is zero.

Static and lexical ***scope rules*** determine the scopes of bindings.

Deep and shallow ***binding rules*** (somewhat confusingly) associate referencing environments with functions that are passed as parameters or return values, or stored in variables.

==========

Scope rules -- static and dynamic

With ***static (lexical) scope rules***, a scope is defined in terms of the physical (lexical) structure of the program.

- Used in most languages today
- Can be figured out by the compiler, without running the program.

Typically choose the closest enclosing binding, as in Algol 60.
Identifier is known throughout scope of declaration (inc. enclosed scopes) unless a new binding for the identifier appears in an enclosed scope.
To resolve a reference, examine the local scope and statically enclosing scopes until a binding is found. More on this under "declaration order" below.

At a given level of nesting, ***modules*** may be used to control visibility.
Names may not be visible outside w/out export.
Names may not be visible inside w/out import.
Rules vary from language to language. Modula-2 and, much more recently, Rust require explicit export from the defining module ***and*** explicit import into the using module. C++, with its namespace mechanism, requires only explicit import: everything in the namespace is implicitly exported.

Modules that require explicit import are said to be ***closed scopes***. Subroutines and nested blocks in most languages are said to be ***open scopes***: identifiers that are not redeclared are automatically inherited from the enclosing scope.

Classes generalize modules (multiple instances, inheritance). These have even more sophisticated (static) scope rules. In particular, methods of a class can generally access members of the class regardless of whether they are nested inside a common lexical (textual) construct.

Bindings created in a subroutine are destroyed at subroutine exit. Modules and classes provide limited scope without limited lifetime. Bindings to variables declared in a module are inactive outside the module, not destroyed. The same sort of effect can be achieved in some languages with `'own'` (Algol term) or `'static'` (C term) variables.

Declaration Order

Two key subtleties of static scoping:

- Does the scope of a binding include the portion of "the scope" before the declaration?
- Does a name have to be declared before use?

Consider the following, in no particular language:

```
const int A = 10
{
    print A
    const A = 12
}
```

What should an interpreter do with this code? Print 10? Print 12? Print an error message?

Some languages (e.g., Pascal, Modula-3, and C#) say the scope of an identifier is its entire block, with a hole in any sub-block in which the identifier is redeclared. Within its block, the identifier must be declared before it is used. So in C#:

```
class C {
    const int A = 10;

    void foo() {
        const int B = A;
            // static semantic error
        const int A = 20;
    }
```

Language rules say the second declaration of A covers all of `foo`, so the declaration of B refers to A before it is declared. Where should you report the error? The mcs compiler complains at the second declaration of A:

```
A local variable `N' cannot be used before it is declared.
Consider renaming the local variable when it hides the member `C.A'.
```

Or this in Pascal:

```
const foo = 10;
...
```

```
procedure P1;
    ...
    procedure P2;
    var A : integer;
    begin
        ...
        A := foo;    {illegal, because of dec. below!}
        ...
    end {P2};
    ...
    procedure foo;
```

Clearly the programmer meant to use the outer `foo`.

There can be arbitrary distance between the (illegal) use and the inner declaration.

Ada, C, C++, Java, and others say scope extends from a declaration to the end of the current block.

C++ and Java dispense with declare-before-use for members, but not locals. Java dispenses with it for classes.
Modula-3 dispenses with it entirely.
Python dispenses with declarations: a variable is local iff written.

Modula-3 and Python share the Pascal/C# whose-scope rule, but don't require declare-before-use, so they don't have the gotchas above.

In OCaml, `let` introduces a name into the expression that follows `in` (or into the global scope, once you hit `;;`). If you want the name visible within the definition (for recursive functions), use `let rec`. If you need mutually recursive functions, use `let rec... and ...`

Declarations v. definitions

Former introduces a name; latter fully describes the named thing.

Declarations that are ***not*** definitions are useful for

- getting around declare-before-use for recursive definitions
  C, C++, Ada, Scheme, ...
- information hiding
  most OO and module-based languages

Dynamic Scope

Where static scope rules determine bindings based on lexical nesting, ***dynamic scope rules*** do it based on execution history—order of subroutine calls and scope entry. To resolve a reference, we use the most recent, active binding elaborated at run time.
Appears in several early languages (early LISP in particular) and in environment variables of modern shell languages.

example (in no particular language)

```
int a
proc first:
    a := 1
proc second:
    int a
    first()

a := 2;  second();  write(a)
```

With static scope, prints a 1. With dynamic scope, prints a 2.
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".

==========

Binding Rules

Recall that the ***referencing environment*** of a statement at run time is the set of active bindings. A referencing environment corresponds to a collection of scopes that are examined (in order) to find a binding.

Scope rules determine that collection and its order.

Binding rules determine which instance of a scope should be used to resolve references when calling a subroutine that was passed as a parameter, returned from a function, or stored in a variable.

That is, they govern the binding of referencing environments to ***formal subroutines***.

With ***shallow binding***, the nonlocal referencing environment of a subroutine is the referencing environment in force at the time it (the subroutine) is called. Original Lisp worked this way by default.

With ***deep binding***, the nonlocal referencing environment of a subroutine is the referencing environment in force when creating a reference to the subroutine at run time -- when passing it to or returning it from some other subroutine, or when storing a reference to it in a pointer.

> [*Aside: "subroutine" v "function" v "procedure" v "method*
> Subroutine is the general term.
> A function is a subroutine that returns a value.
> A procedure is a subroutine that does *not* return a value
> (it is executed solely for its side effects).
> A method is a subroutine that belongs to a class, with special scope rules, access to instance variables, and usually dynamic choice among subclass instances.
> I am sometimes sloppy about using these; bear with me.]

For subroutines passed as parameters, this environment captured by deep binding is the same as would be extant if the procedure were actually called at the point where the reference was created. When the reference is passed or saved, this referencing environment is passed or saved as well. When the subroutine is eventually called (through the reference), this saved referencing environment is restored. The original Lisp made this behavior available when desired; it's the default in most modern languages.

A subroutine reference together with its bundled referencing environment is called a ***subroutine closure***. There are several possible implementations; the simplest is code address plus a copy of the static link.

First and second-class subroutines

first: can pass, return, store; second: can pass, but not return or store

Why not return or store?
Limited v. unlimited extent.

Example: (* OCaml *)

```
let plus_n n = fun k -> n + k;;
let plus_3 = plus_n 3;;
let apply_to_2 f = f 2;;

apply_to_2 plus3                => 5
```

Here the n inside `plus_n` needs to have a lifetime that extends beyond the call inside `plus_3`.

Note 1: The difference between deep and shallow binding is not apparent unless you pass subroutines as parameters, return them from functions, or store references to them in variables. Binding rules are therefore irrelevant in languages that lack formal subroutines: you don't need closures if you don't have formal subroutines.

Note 2: To the best of my knowledge, no language with static (lexical) scope rules has used shallow binding: it's possible to figure out what that combination would do, but it really doesn't make sense. Some languages with dynamic scope rules (e.g., Snobol) offered only shallow binding; others (eg. early Lisp) offered both. Hence, the issues are separable.

Note 3: In a language with lexical scope, the difference (if anybody cared) would only be noticeable for non-local references, that is, references which are neither local nor global. Binding rules would have no relevance to (lexical) local/global references since all local references are always bound to the currently executing instance and there is only one instance of the main program containing the global variables. Binding rules are therefore irrelevant in languages such as C, which lack nested subroutines, or Modula-2, which allow only outermost subroutines to be passed as parameters, and would also be irrelevant in a language with nested subroutines but no recursion (I'm not aware of any like that). So closures are trivial with static scope and no nested subroutines.

Example of why deep binding matters for static scope (in OCaml): Scan a list. Return the sum of element k and first negative element (if any) prior to k:

```
let foo l k =
  let rec helper l f i b =
    match l with
    | [] -> raise (Failure "list too short")
    | h :: t ->
        if i = k then f h
        else if (b && h < 0) then helper t (( + ) h) (i + 1) false
                  else helper t f (i + 1) b in
      helper l (fun x -> x) 0 true;;
```

This captures the "right" h in `foo [1; -3; 2; -4; 5] 4;;`

Note that the OCaml implementation doesn't use a straightforward application of static links, because of tail recursion optimization.
The equivalent code in Python would:

```
def f(l, k):
    def helper (l, f, i, b):
        if l == []:
            return "list too short"
        elif i == k:
            return f(l[0])
        elif b and l[0] < 0:
            h = l[0]
            return helper(l[1:], (lambda x: h + x), i+1, False)
        else:
            return helper(l[1:], f, i+1, b)

    return helper(l, (lambda x: x), 0, True)

l = [1, -3, 2, -4, 5]
print(f(l, 4))
```

We'll return to implementation techniques for scope and binding rules in chapter 9.

NB: object-oriented languages without first-class subroutines can get some of the same effect using ***object closures***: create an object whose fields hold values that would have been in the referencing environment of a subroutine closure; pass the object to somebody; let them invoke one of its methods. The operator() mechanism of C++ makes this look like ordinary subroutine invocation.

Lambda expressions

Function ***aggregates*** -- in-line built-up values of functional type.
Familiar to users of functional languages.
Increasingly common in imperative languages as well.
Require unlimited extent—as in Ruby, C#, and Scala—or explicit management of ownership—as in Rust—to really work well. Can still be useful without it, though, as in Java 8 and C++'11.

Here's a C++ example:

```
auto plusx(int x) {
    return [x](int y){ return x + y; };
}
```

This captures x by value. Capturing by reference `[&x]` would not work here, due to limited extent.

==========
Other concepts related to naming

Aliases: using more than one name for the same thing
Problems:
  potentially confusing
  inhibit code improvement (e.g., promotion to registers)

What are aliases good for?
 * linked data structures
 x space saving -- modern data allocation methods are better
 x multiple representations -- unions are better

Aliases sometime arise in parameter passing as an unfortunate side effect.
Euclid scope rules are designed to prevent this.

Overloading: using the same name for multiple things

Some overloading happens in almost all languages
- integer + v. real +
- built in I/O operations in some languages
- function return in Pascal

Some languages get into overloading in a big way
- Ada
- C++

```
overload norm;
    int norm (int a) { return a > 0 ? a : –a; )
    complex norm (complex c ) { // ...
```

overloading is also known as “ad hoc polymorphism.”

(True) Polymorphism

Means, literally, “having many forms.”
In practice, it means “applicable to many types.”
There are several different variants.

Simplest is ad hoc polymorphism, which really doesn’t deserve the name.

Subtype polymorphism in OO languages allows code to do “the right thing” to parameters of different types in the same type hierarchy by calling the virtual function appropriate to the concrete type of the actual parameter.

Explicit parametric polymorphism (generics)

You specify type parameters when you declare or use the generic.

Templates in C++ are an example of this:

```
typedef set<string>::const_iterator string_handle_t;
set<string> string_map;
...
pair<string_handle_t, bool> p = string_map.insert(ident);
```

Here `pair.first` is the string we inserted; `pair.second` is true iff it wasn’t there before

Implemented via macro expansion in C++ v1; built-in in Standard C++.
Similar mechanisms in Clu, Ada, Java, C#, Scala, ...
May be implemented with

- a single copy of the code that always manipulates references
- a separate copy of the code for every (set of similar) type(s)

Implicit (true) parametric polymorphism

You don't have to specify the type(s) for which code works; the language implementation figures it out and won't let you perform operations on items that don't support them. Functional languages generally support true parametric polymorphism, either in the runtime system (Lisp and its descendants) or in the compiler (ML and its descendants, inc. Haskell).

More on polymorphism in Chapter 7.