Naming, scoping, binding, and related notions (October 5 and 7, 2020)

==========

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 a 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 "object," 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 things using symbolic identifiers rather than values, addresses, etc. Things 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.

Let's call anything that is represented by bits in memory as an ***object***. (This is ***not*** using the term in the OO programming sense.)

The ***lifetime*** of an object 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 object 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 object
- creation of binding
- uses of name that is bound to object
- (temporary) deactivation (hiding) of binding
- reactivation of binding
- destruction of binding
- destruction of object

If an object outlives its binding it's ***garbage***.
If a binding outlives its object 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 on the rules that determine scope in the following lecture.)

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.

---

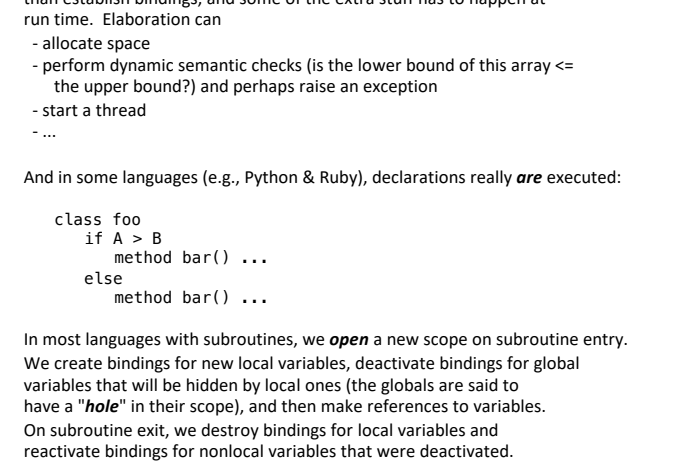
Storage Management -- for objects with various lifetimes.

- 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

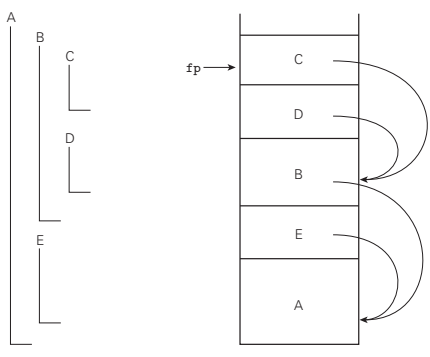
- Heap (chap. 7) for
  - dynamic allocation

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.

Next lecture: static and lexical ***scope rules***, which determine the scopes of bindings.

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