Chapter 3::
Names, Scopes, and Bindings

We have seen various bindings:

- Ruby: type bound to variable during execution
- C++ (inheritance) – Variable bound to type at run time
- C++(non virtual) – variable bound to type at compile time
- C++ parameter – variable bound to space at run time
- Java – integer bound to max length at language definition time
Binding times can vary widely:

- Value of an expression: during execution or during translation (constant expression).

  ```
  const int size = 100;
  const int num = 20;
  const int MAX = size / num;
  int array[MAX];
  ```

- Data type of an identifier: translation time (C++) or execution time (Ruby).

- Maximum number of digits in an integer: language definition time or language implementation time.

- Location of a variable: load or execution time.

- Two general categories of binding: static (prior to execution) and dynamic

- Interpreters will perform most bindings dynamically

- Concern is earliest time when it COULD be bound, not when it actually is

  Possible times
  - Language design
  - Language implementation
  - Program writing time
  - Compile time
  - Link time
  - Load time
  - Execution time - dynamic

  } static
Classes of Binding Times (listed from late to early)

1. Execution Time (Late Binding).
   Variables to their values.
   Variables to a particular storage location (termed dynamic storage allocation).
   - At entry to a subprogram or block.
     Example: formal to actual parameters and formal parameters to actual locations.
   - At arbitrary points during execution.
     Example: variables to values. In some languages, variables to types.
     Consider Prolog - variable type is determined dynamically

2. Load time: globals bound to location

3. Link time: body of external function bound to call instruction

4. Compile time (Early Binding).
   - Bindings chosen by programmer. Variable names, types, names.
   - Bindings chosen by translator.
     Example: particular machine instruction for a statement.
     Example: initial values of variables (if none specified)
     Example: in C declaration defines type but gives no space

5. Language Implementation Time.
   Example: Association of enumerated type values with integers.
   Example: maxint

6. Language designTime - probably the most important binding time.
   Example: structure of language fixed, set of all basic data types, set of statements: syntax and semantics fixed, predefined types.

---

Binding

- run time
  - value/variable bindings, sizes of strings
  - includes
    - program start-up time
    - procedure entry time
    - block entry time
    - statement execution time
In general, early binding times are associated with greater efficiency.
Later binding times are associated with greater flexibility.
Compiled languages tend to have early binding times.
Interpreted languages tend to have later binding times.

How are the bindings remembered?
Symbol table and environment

- A dictionary or table is used to maintain the identifier/attribute bindings.
- It can be maintained either during translation (symbol table) or execution (environment table) or both.
- Pre-translation entities (keywords) are entered into the initial or default table.
- If both are maintained, the environment can usually dispense with names, keeping track only of locations (names are maintained implicitly).

Example of Environment

So – define an environment

```c
struct People { ... } x;
int f ( int a) {
    double y;
    int x;
    ...  Point 1
}
int main () {
    double a;
    ...  Point 2
}
```

Point 1:
- struct People → type,
- x→int,
- f → func,  a → int,
- y → double

Point 2:
- struct People → type,
- x → struct People,
- f → func,
- a → double,
- main → func
Scope

- The scope of a declaration is the region of the program to which the bindings established by the declaration apply.
- Informally - Scope of a variable: range of statements in which the variable is visible
- A variable is visible in a statement if it can be referenced in that statement. (Scope holes caused by new declarations)
- In a block-structured language, the scope is typically the code from the end of the declaration to the end of the "block" (indicated by braces {...} in C) in which the declaration occurs.
- Scope can extend backwards to the beginning of the block in certain cases (class declarations in Java and C++).

Scope Rules

- A scope is a program section of maximal size in which no bindings change, or at least in which no re-declarations are permitted
- In most languages with subroutines, we OPEN a new scope on subroutine entry:
  - create bindings for new local variables,
  - deactivate bindings for global variables that are re-declared (these variable are said to have a "hole" in their scope)
  - make references to variables
Lexical vs. dynamic scope

- Scope is maintained by the properties of the lookup operation in the symbol table or environment.
- If scope is managed statically (prior to execution), the language is said to have **static or lexical scope** ("lexical" because it follows the layout of the code in the file).
- If scope is managed directly during execution, then the language is said to have **dynamic scope**.
- It is possible to maintain lexical scope during execution - via static links in the call stack.

Java scope example

```java
public class Test {
    public static int x = 2;
    public static void q(String[] args) {
        int x = 0;
        System.out.println(x);
    }

    public static void f() {
        System.out.println(x);
        q();
    }

    public static void main(String[] args) {
        int x = 3;
        f();
    }
}
```

- This prints 2 0, but under dynamic scope it would print 3 0 (the most recent declaration of x in the execution path is found).
Dynamic scope evaluated

- Almost all languages use lexical scope (including Ruby - every class or class has its own local scope). Why do you think that is true?
- With dynamic scope the meaning of a variable cannot be known until execution time, thus there cannot be any static checking.
- Originally used in Lisp. Scheme could still use it, but doesn't. Some languages still use it: VBScript, Javascript, Perl (older versions).
- Lisp inventor (McCarthy) now calls dynamic scope a bug.
- Still useful as a pedagogical tool to understand the workings of scope.

Symbol table construction

- Symbol table is constructed as declarations are encountered (via insert operation).
- Lookups occur as names are encountered
- In lexical scope, lookups occur either as names are encountered in symbol table to that point (declaration before use—C), or all lookups are delayed until after the symbol table is fully constructed and then performed (Java class—scope applies backwards to beginning of class).
- In dynamic scope, need links to tell you which declarations to use
- If a different scope applies (like Ruby code blocks), the environment needs to be passed.
Overloading

- Overloading is a property of symbol tables that allows them to successfully handle declarations that use the same name within the same scope.
- It is the job of the symbol table to pick the correct choice from among the declarations for the same name in the same scope. This is called overload resolution.
- Overloading typically applies only to functions or methods.
- Overloading (static decision) is different from dynamic binding in an OO language.

An example in Java:

```java
public class Overload {
    public static int max(int x, int y)
    { return x > y ? x : y; }
    public static double max(double x, double y)
    { return x > y ? x : y; }
    public static int max(int x, int y, int z)
    { return max(max(x,y),z); }
    public static void main(String[] args)
    { System.out.println(max(1,2));
      System.out.println(max(1,2,3));
      System.out.println(max(4,1.3));
    }
}
```

- Adding more `max` functions that mix `double` and `int` parameters is ok. But adding ones that mix `double` and `int` return values is not!
Allocation of space for variables

- Can be constructed entirely statically (Fortran): all variables and functions have fixed locations for the duration of execution.
- Can also be entirely dynamic: functional languages like Scheme and ML.
- Names of constants may be discarded - stored in code itself.
- Most languages use a mix: C, C++, Java, Ada.
- Space consists of three components:
  - A fixed area for static allocation
  - A stack area for lifo allocation (usually the processor stack)
  - A "heap" area for on-demand dynamic allocation (with or without garbage collection)

Typical environment organization
Lifetime and Storage Management

- Heap for dynamic allocation

![Heap diagram]

**Figure 3.2 Fragmentation.** The shaded blocks are in use; the clear blocks are free. Cross-hatched space at the ends of in-use blocks represent internal fragmentation. The discontinuous free blocks indicate external fragmentation. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

The Runtime Stack

- Used for:
  - Procedure/function/method calls
  - temporaries associated with function call
  - local variables

  Runtime stack -overflows in infinite recursion

- Temporaries: intermediate results that cannot be kept in registers.
- Procedure call: parameters and return values
- Local variables: part of calls, but can be considered independently, showing LIFO behavior for nested scopes (next slide).
Typical Activation Record for a Language with Stack-Dynamic Local Variables

<table>
<thead>
<tr>
<th>Local variables</th>
<th>Parameters</th>
<th>Dynamic link</th>
<th>Return address</th>
</tr>
</thead>
</table>

Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

- The activation record format is static, but its size may be dynamic
- The dynamic link points to the top of an instance of the activation record of the caller
- An activation record instance (ARI) is dynamically created when a subprogram is called
- Run-time stack
An Example: C Function

```c
void sub(float total, int part)
{
    int list[5];
    float sum;
    ...
}
```

An Example Without Recursion

```c
void A(int x) {
    int y;
    ...
    C(y);
    ...
}
void B(float r) {
    int s, t;
    ...
    A(s);
    ...
}
void C(int q) {
    ...
}
void main() {
    float p;
    ...
    B(p);
    ...
}
```

main calls B
B calls A
A calls C
An Example Without Recursion

The environment would be a pointer to an activation record in the runtime stack.

Heap Allocation

- In "standard" languages (C, C++, Java) heap allocation requires a special operation: new.
- The stack is still used to represent calls.
- In C/C++, deallocation is typically by hand (destructors), but it is hard to do right.
- Java uses a garbage collector that periodically sweeps the heap looking for data that cannot be accessed any more by the program and adding it back to free space.
Lifetime

- The **lifetime** of a program entity is the duration of its allocation in the environment.
- Allocation is **static** when the lifetime is the duration of the entire program execution.
- Lifetime is related to but not identical to scope. With **scope holes**, lifetime can extend to regions of the program where the program entity is not accessible. It is there, but you can’t access it.

Aliases

- An **alias** occurs when the same object is bound to two different names at the same time.
  
  *Give an example in C++ where this occurs.*
The Meaning of Names within a Scope

- **Aliasing**
  - What are aliases good for? (consider uses of FORTRAN equivalence)
    - space saving - modern data allocation methods are better
    - multiple representations - unions are better
    - pointers - legit
  - Also, aliases arise in parameter passing as an unfortunate effect
  - Why do we care about aliases?
    - confusion
    - interference
    - program optimizations (such as common subexpression elimination) are hampered.

Dangling References, and Garbage

- **A dangling reference** is a location that has been deallocated from the environment, but is still accessible within the program. Dangling references are impossible in a garbage-collected environment with no direct access to addresses.
- **Garbage** is memory that is still allocated in the environment but has become inaccessible to the program. Garbage can be a problem in a non-garbage collected environment, but is much less serious than dangling references.
Lifetime and Storage Management

- Contents of a stack frame
  - arguments and return values
  - local variables
  - temporaries
  - bookkeeping (saved registers, line number static link, etc.)

- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time

---

Figure 3.1 Stack-based allocation of space for subroutines. We assume here that subroutines have been called as shown in the upper right. In particular, B has called itself once, recursively, before calling C. If D returns and C calls E, E's trace (subroutine stack) will use the same space previously used for D's frame. At any given time, the stack pointer (sp) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (fp) register points to a known location within the frame of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

---

Copyright © 2009 Elsevier
Lifetime and Storage Management

- Maintenance of stack is responsibility of calling sequence and subroutine prolog and epilog
  - space (code to manage) is saved by putting as much in the prolog and epilog as possible
  - time may be saved by
    - putting stuff in the caller instead or
    - combining what's known in both places (inter-procedural optimization)

Scope Rules

- On subroutine exit:
  - destroy bindings for local variables
  - reactivate bindings for global variables that were deactivated
- Algol 68:
  - ELABORATION = process of creating bindings when entering a scope
- Ada (re-popularized the term elaboration):
  - storage may be allocated, tasks started, even exceptions propagated as a result of the elaboration of declarations
• **closed scope**: names must be explicitly imported
• **open scope**: no explicit imports
• **selectively open**: `A.foo` works, but can reference as `foo` if import. (like using namespace `std` in C++)

---

**Scope Rules**

• Note that the bindings created in a subroutine are destroyed at subroutine exit
  – The modules of Modula, Ada, etc., give you closed scopes without the 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 many languages with `own` (Algol term) or `static` (C term) variables (see Figure 3.5)
Scope Rules

- Access to non-local variables **STATIC LINKS**
  - Each frame points 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

C++ - nested scopes. Each scope could have an activation record.

```c++
int x = 10;
class Y
  string x;
  foo ()
  {  Student x;
      doit ('a');
  }
  doit (char x)
  {
    for (int x=0; x < 10; x++)  <= What is known?
  }
main
  Y y;
  y.foo();
```
Scope Rules

The key idea in **static scope rules** is that bindings are defined by the physical (lexical) structure of the program. **At compile time, the system figures out how many static links it has to follow, making it more efficient.**

- You can use a display to make it more efficient. **Display** - a vector of pointers to currently active static chain frames on the runtime stack.

Figure 3.5  Static chains. Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, B, D, and C then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the frame pointer; it can find local objects of the surrounding scope B by dereferencing its static chain once and then applying an offset. It can find local objects in B's surrounding scope A by dereferencing its static chain twice and then applying an offset.

Copyright © 2009 Elsevier
Scope Rules

- **With dynamic scope rules,**
  - To resolve a reference, we use the most recent, active binding made at run time

**polymorphism** is a programming language feature that allows values of different data types to be handled using a uniform interface

- **polymorphic function**: generalized parameter type (often thru inheritance or interface)
- **ad hoc polymorphism**: overloading
- **generic function**: a syntactic template that can be instantiated in more than one way at compile time
Static scope requires following static chains. May want to save time with a display

Example

Display at /*8*/:

Red is dynamic pointers. Blue dotted is static pointers.

Lexical Scoping with Recursion

main{
    a, b, c: integer;/*1*/
    procedure P0;/*2*/
    c: integer;
    if (...) /*3*/ P0()
    else return;
    ...
}/end P
    ... P0;/*4*/......
}/end main

After recursive call of P0 within P0 at /*5*/; this is stack at /*2*/ in recursive call
Binding of Referencing Environments

● Accessing variables with dynamic scope:
  – (1) keep a stack (association list) of all active variables
    • When you need to find a variable, hunt down from top of stack
    • This is equivalent to searching the activation records on the dynamic chain

Binding of Referencing Environments

● Accessing variables with dynamic scope:
  – (2) keep a central table with one slot for every variable name
    • If names cannot be created at run time, the table layout (and the location of every slot) can be fixed at compile time
    • Otherwise, you'll need a hash function or some method to do lookup
    • Every subroutine changes the table entries for its locals at entry and exit.
Central Reference Table

- Can't use <base,offset> addressing of display to implement dynamic scoping because dynamic chain is NOT FIXED LENGTH
- Try to minimize cost of run-time variable lookup
- Run-time access to variables is indirect through this hash table, 1 entry per active identifier name

Central Reference Table

- 1 entry per distinct identifier name plus active/inactive flag
  - If active flag on, entry contains variable's address
- Procedure prologue initializes the table entries for local variables of this procedure (each entry is really a stack)
- Procedure epilogue pops entries for local variables from the table
Binding of Referencing Environments

- (1 association list) gives you slow access but fast calls
- (2 central table) gives you slow calls but fast access
- In effect, 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 are more complicated
Binding of Referencing Environments

- **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

Conclusions

- The morals of the story:
  - Language features can be surprisingly subtle
  - Designing languages to make life easier for the compiler writer *can* be a GOOD THING
  - Most of the languages that are easy to understand are easy to compile, and vice versa
Conclusions

- A language that is easy to compile often leads to
  - more good compilers on more machines
  - better (faster) code
  - fewer compiler bugs
  - smaller, cheaper, faster compilers
  - better diagnostics