Chapter 3. Names, Scopes, and Bindings

- A ***name*** is a ***mnemonic character string*** used to represent something else.

  (ex) identifiers (alphanumeric tokens), symbols such as + or :=

- ***Names*** allow us to refer to ***variables***, ***constants***, ***operations***, ***types***, and so on using symbolic identifiers rather than low-level concepts like addresses.

- ***Abstraction*** is a process by which the programmer ***associates a name with potentially complicated program fragment***.

- By hiding irrelevant details, ***abstraction reduces conceptual complexity***, making it possible for the programmer to ***focus on manageable subset of the program text at any particular time***.

  (ex) - Subroutines are control abstractions.
  - Classes are data abstractions.

3.1 The Notion of Binding Time

- A ***binding*** is an ***association between two things***, such as a name and the thing it names (i.e. ***mapping symbol references to intended object***).

- ***Binding time*** is the time at which ***a binding is created*** or, more generally, the time at which ***any implementation decision is made***.

- There are ***many different times*** at which ***decisions may be bound***:

  (1) ***Language design (or definition) time***

  - In most languages, the control flow constructs, the set of fundamental (primitive) type, the available constructors for complex types, and many other aspects of language semantics are chosen when the language is designed.

    (ex) - * is bound to the multiplication,
    - type (: integer, real, decimal, etc.),
    - operator (*, /, +, etc.),
    - many choose that 100 is in decimal.

  (2) ***Language implementation time***

  - Most language manuals leave a variety of issues to the discretion of the language

implementor.

(ex) - precision (# of bits) of the fundamental types,
- 48 bit one's complement + 1 to represent negative number
- range of possible integer value for Pascal
- the organization and maximum size of stack an heap,
- handling of run-time exceptions,
- div means division for integer,
- / means division for real

(3) ***Program writing time***

- Programmers choose algorithms, data structures, and names.

(4) ***Compile time***

- Compilers choose the mapping of high-level constructs to machine code, including the layout of statically defined data in memory.

(ex) Variable in Fortran program is bound to a real type,
* means integer and real multiplication and assigned to an integer variable, A

```
Integer A, B
   …
C = 55
A = B * C (Note: C is not declared but considered as a real type)
```

(5) ***Link time***

- Since most compilers support ***separate compilation***- compiling different modules of a program at different time- and depend on the availability of a library of standard subroutines, a program is usually not complete until the various modules are joined together by a linker.

(ex) Y := SQRT(A); will call Function SQRT(X) (Note: A call statement link the subroutine.)

(6) ***Load time***

- Load time refers to the point at which the ***operating system loads the program into memory*** so that it can run. In primitive O.S.s, the choice of machine addresses for object within the program was not finalized until load time. In modern O.S.s, virtual addresses are chosen at link time and physical addresses can actually change at run time.

(ex) - The actual memory cell is allocated for a variable.
- Load each subprograms to physical address.
- The processor's memory management hardware translates virtual addresses into physical addresses during each individual instructions at run time.

(7) ***Run time***

- A very broad term that covers the entire span from the beginning to the end of execution. Run time subsumes program start-up time, module entry time, elaboration time (the point at which a declaration is first "seen"), subroutine call time, block entry time, and statement execution time.

(ex) Bindings of values to variables (i.e. A=10)

- Things bound ***before run time*** are referred ***static***, and ***at run time*** are referred ***dynamic***.

- ***Compiler-based language*** implementations tend to be ***more efficient*** than interpreter-based implementations because ***they make earlier decisions***.

- Some languages are ***difficult to compile*** because their definitions require ***fundamental decisions to be postponed until run time***, generally in order ***to increase the flexibility or expressiveness*** of the language.

(ex) Smalltalk delays all type checking until run time.

3.2 Object Lifetime and Storage Management

- The period of time between the creation and the destruction of a name-to-object binding is called the ***binding's lifetime***.

- The time between the creation and destruction of an object is the ***object's lifetime***.

- An object may retain its value and the potential to be accessed even when a given name can no longer be used to access it.

(Note) If ***object outlives binding***, it is ***garbage***.
Garbage is an object that cannot be reached through a reference.

(ex) When a variable is passed to a subroutine by reference, for example (as it typically is in **Fortran**, with '**var**' parameters in Pascal, or with '**&**' parameters in C++), the binding between the parameter name and the variable that was passed has a lifetime shorter than that of the variable itself.

- It is also possible for a name-to-object binding to have a lifetime longer than that of the object.

(ex) If an object created via the C++ new operator is passed as a & parameter and then deallocated before the subroutine returns (i.e. "***dangling reference***").

(Note) If ***binding outlives object*** (i.e. a binding to an object that is no longer live), it is a ***dangling reference***.

- Object lifetimes generally correspond to one of the 3 strong allocation mechanism:
  (1) ***Static*** objects are given an absolute address.
  (2) ***Stack*** objects are allocated and deallocated in L.I.F.O. order
  (3) ***Heap*** objects may be allocated and deallocated at arbitrary times.

3.2.1 Static Allocation

- Examples of static object:
  (1) global variables
  (2) the instructions that constitute a program's machine language translation
  (3) variables that are local to a single subroutine, but retain their values from one invocation to the next (ex) static variable in C++
  (4) numeric and string-valued constant literals
  (ex) A =B / 14.7
  printf("hello, world\n")
  (5) tables that are produced by compiler and used by runtime support routines
  (ex) symbol table

- Statically allocated objects ***whose value should not change during program execution*** are often ***allocated in protected read-only memory***.

- ***Local variables*** are ***created when their subroutine is called***, and ***destroyed when it returns***. ***Recursion was not originally supported in Fortran (added in Fortran 90)***. As a result, there can never be more than one invocation of a subroutine active in an older Fortran program at any given time, and a compiler may ***choose to use static allocation for local variables***.

- ***Manifest constant*** (or ***compiler-time constants***) can always be ***allocated statically***, even if they are local to a recursive subroutine: ***multiple instances can share the same location***.

- ***Elaboration-time constants***, when local to a recursive subroutine, must be allocated on the stack.

3.2.2 Stack-Based Allocation

- If a language ***permits recursion***, static allocation of local variable is no longer an option.

- The natural nesting of subroutine calls make it easy ***to allocate space for locals on a stack***.

- See Figure3.1 Stack-based allocation of space for subroutines (p121 of the textbook).

- Each instance of a subroutine at run time has its own ***frame*** (or ***activation record***) on the stack, containing ***arguments*** and ***return values***, ***local variables***, ***temporaries***, and ***book keeping information***. ***Arguments to be passed to subsequent routines lie at the top*** of the frame.

- The organization of the remaining information is ***implementation-dependent***.

(Note) The contents of a activation record:

(1) Return Address (RA)
(2) Dynamic Chain Pointer (DCP)–who called this module (i.e. dynamic father)
(3) Static Chain Pointer (SCP)–a pointer to the module environment in which this module was created (i.e. static father)
(4) Stack Top pointer–a pointer to the next free location in the stack
(5) Reference–location for variables, labels, etc. in subprograms
(6) Parameter Information

- Maintenance of the stack is the responsibility of subroutine "***calling sequence***"– ***the code executed by the caller immediately before and after the call***.

- The ***offsets of objects*** within a frame usually can be ***statically determined***. The compiler can arrange for a particular register, known as the ***frame pointer to always point to a known location*** within the frame of the current subroutine. A local variable within the current frame,

or an argument near the top of the calling frame can be ***accessed by adding a predetermined offset to the value in the frame pointer***.

3.2.3 Heap- Based Allocation

- A ***heap*** is a region of storage in which ***subblocks*** can be ***allocated and deallocated at arbitrary times***.

- Heaps are required for the ***dynamically allocated pieces*** of linked data structures, and for objects like fully general character strings, lists, and sets, where ***size may change*** as a result of an assignment statement or other updated operation.

- The principle concerns in managing a heap are ***speed*** and ***space***. There are ***tradeoffs between them***.

- Two issues in space concern (See Figure 3.2 Fragmentation, p122 of the textbook):

  (1) ***Internal fragmentation*** occurs when a storage-management algorithm allocates a block that is larger than required to hold a given object; ***the extra space is then unused***.

  (2) ***External fragmentation*** occurs when the blocks that have been assigned to active objects are scattered through the heap in such a way that the remaining, ***unused space is composed of multiple blocks***: there may be quite a lot of free space, but ***no one piece of it may be large enough*** to satisfy some future request.

3.2.4 Garbage collection

- Many languages specify that objects are to be deallocated implicitly when it is no longer possible to reach them from any program variable.

- The run-time library for such a language must then provide a ***garbage collection*** mechanism ***to identify and reclaim reachable objects***.

- Most functional and scripting languages require garbage collection, as do many more recent imperative languages, including Modula-3, Java, and C#.

- The traditional arguments in favor of explicit deallocation are ***implementation simplicity*** and ***execution speed***.

- The argument in favor of automatic garbage collection, however, is compelling: manual deallocation errors are among the most common and costly bugs in real-world programs.

(c.f.) ***dangling reference*** and ***leaking memory*** problems

- ***Deallocation errors*** are notoriously ***difficult to identify and fix***.

- Over time, both language designers and programmers have increasingly come to ***consider automatic garbage collection an essential feature***.

3.3 Scope Rules

- The ***textual region of the program*** in which ***a binding is active*** is its ***scope***.

- In ***most modern languages***, the ***scope of a binding*** is determined ***statically*** (i.e. at compile time). (ex) C, Fortran, …

- Other languages, including APL, Snobol, and LISP, are ***dynamically scoped***: their ***bindings depend on the flow execution*** (i.e. calling sequence of subprograms) at run time.

3.3.1 Static Scoping

- In a language with ***static scoping***, the ***binding*** between names and objects can be ***determined at compile time*** by examining the text of the program.

- Typically, the ***current binding*** for a given name ***is found in the matching declaration whose block most closely surrounds a given point in the program***.

(ex)

(Note) (1) A declaration for a variable effectively hide any declaration of a variable with the same name in a larger enclosing scope.
(2) Non-local visible

- ***Early versions of Basic***: there was only ***a single, global scope***. There were no explicit declarations; variables were declared implicitly by virtue of being used.

- ***Fortran distinguishes between global and local variables***. The scope of a local variable is limited to the subroutine in which it appears. If a variable is not declared, it is assumed to local to the current subroutine and to be of type integer or real. Global variables in Fortran may be partitioned into common blocks, which are then "imported" by subroutines.

- ***Most compiled languages*** such as C, Pascal, Ada, Algol 6, etc. employ ***static scope rules***.

3.3.2 Nested Subroutines

- Algol-style nesting gives rise to the ***closest nested scope rule*** for bindings from names to object: a ***name*** that is introduced in a declaration is ***known in the scope in which it is declared***, and ***in each internally nested scope***, ***unless it is hidden by*** another declaration of the ***same name*** in one or more nested scopes.

- To find the object corresponding to a given use of a name, we ***look for a declaration with that name in the current, innermost scope***. If there is one, it defines the active binding for the name. Otherwise, we ***look for a declaration in the immediately surrounding scopes, until we reach the outer nesting level of the program, where global objects are declared***. If no declaration is found at any level, then the program is in error.

(ex) Figure 3.4 Example of nested subroutines, shown in pseudocode (p129 of the textbook)

- A name-to-object binding that is hidden by a nested declaration of the same name is said to ***have a hole in its scope***.

- ***In most languages the object whose name is hidden is inaccessible in the nested scope***. Some languages allow the programmer to ***access the outer meaning of a name*** by applying ***qualifier*** or ***scope resolution operator***.

(ex) In Ada, My_proc. X refers to the declaration of X in subroutine My_proc.

(ex) In C++, ::X refers to a global declaration of X, regardless of whether the current subroutine also has X.

- The compiler can arrange for a frame pointer register. ***Using this register as a base for***

displacement addressing, target code can access objects within the current subroutine.

- To find objects in lexically surrounding subroutines, we need a way to find frames corresponding to those scopes at run time. Since a nested subroutine may call a routine in an outer scope, the order of stack frames at run time may not necessarily correspond to the order of lexical nesting. The simplest way to find the frames of surrounding scope is to maintain a ***static link*** in each frame that points to the ***parent*** frame.

(ex) Static chain:

(Note) If a subroutine is declared at the outmost nesting level of the program, then its frame will have a ***null*** static link.

3.3.3 Declaration Order

- Suppose an object X is declared somewhere within block B. ***Does the scope of X include the portion of B before the declaration***, and if so can X actually be used in that portion of the code?

- Pascal says that (1) names must be ***declared before*** they are ***used*** and
(2) the scope of a declaration is the ***entire surrounding block***.

(ex) const N= 10;
…
procedure foo;
const
M=N; (* static semantic error! *)
…
N=20; (* local constant declaration; hides the outer N *)

(Note) The error has the potential to be ***highly confusing*** particularly ***if the programmer meant to use the outer N***.

(ex) const N=10;
…
procedure foo;
const
M=N; (* static semantic error! *)
var

```
A: array [1..M] of integer;
N: real;   (* hiding declaration *)
```

(Note) N used before declaration and N is not a constant.

- Most ***Pascal successor*** (and some dialects of Pascal itself) specify that the scope of an identifier is the portion of the block (in which it is declared) ***from the declaration to end (excluding holes)***. If the above program fragment had been written in Ada, for example, or in C, C++, or Java, no semantic errors would be reported.
  C++ and Java further relax the rules by dispensing the define-before-use requirement in many cases.

- While ***C#*** echoes Java in requiring ***declaration before use for local variables*** (but ***not for classes and members***), it returns to the Pascal notion of whole-block scope.

(ex) ***Invalid*** in C#,

```
Class A {
    const int N=10;
    void foo () {
        const int M=N; //uses inner N before it is declared
        const int N=20;
```

- Module-3 uses the ***simplest approach*** which says that the scope of a declaration is the ***entire block*** in which it appears (***minus any holes*** created by nested declarations), and that the ***order of declarations doesn't matter***.

- Python takes the "***whole block***" scope rule one step further by ***dispensing with variable declaration altogether*** (i.e. ***No variable declaration*** is needed). The local variables of subroutine S are precisely those variables that are written by some statement in the (static) body of S.

- Declaration and Definitions: Recursive types and subroutines introduce a problem for languages that require names to be declared before they can be used (i.e. how can two declarations each appear before the other?). C and C++ handle the problem by distinguishing between the "***declaration***" of an object and its "***definition***".
  introduces a name and indicates its scope

(ex)
```
struct manager;                          / * declaration only * /

struct employee {
    struct manager * boss;
    struct employee  * next_employee;
      …
};
```

```
struct manager {                                  / * definition * /
        struct employee * first_employee;
         …
};
(ex) void list_tail (follow_set fs);              / * declaration only * /

void list (follow_set fs)
{
        switch (input_token) {
              case id: match (id); list_tail(fs);
        …
}

void list_tail (follow_set fs)                    / * definition * /
{
        switch (input_token) {
        case comma: match (comma); list (fs) ;
       …
}
```

- Nested Blocks:

  In many languages, including Algol 60, C89, and ADA, ***local variables*** can be declared ***not only at the beginning of any subroutine, but also at the top of any begin … end ({…}) block***. Other languages, including Algol 68, C99, and all of C's descendants, are ***even more flexible, allowing declarations whenever a statement may appear***.

```
(ex) {    int temp= a;
          a=b;
          b= temp;   }
```

(Note) ***keeping the declaration of temp lexically adjacent to the code that uses it*** makes the program ***easier to read***, and ***eliminates any possibility that this code will interfere with another variable named temp***

3.3.4 Modules

- A module allows a collection of objects—subroutines, variables, types, and so on—to be encapsulated in such a way that

  (1) objects ***inside are visible to each other***, but
  (2) objects on the ***inside are not visible on the outside unless explicitly exported***, and
  (3) (in many languages) objects ***outside are not visible on the inside unless explicitly imported***.

(Note) These rules ***affect only the visibility*** of objects; they ***do not affect their lifetime***.

(ex) Fig. 3.6 Stack abstraction in Moduls-2 (3rd edition of the textbook)
Fig. 3.6 Pseudorandom number generator module in C++ (p137 of the textbook)

- Several languages, including Ada, Java, and Perl, use the term ***"package"*** instead of module. Other, including C++, C#, and PHR, uses ***"namespace"***.

- ***Module facilitate the construction of abstractions*** by allowing data to be made private to the subroutines that use them.

- As an extension of the module-as-type approach to data abstraction, many languages now provide a class constructor for "object-oriented programming". ***Classes can be thought of as module types that have been augmented with an "inheritance" mechanism***.

3.3.6 Dynamic Scoping

- In a language with ***dynamic scoping***, the bindings between names and objects depend on the flow of control at run time, and in particular on the ***order in which subroutines are called***.

(ex) Figure 3.9 Static versus dynamic scoping (p143 of the textbook)