

Names, Bindings, and Scopes

5.1 Introduction 198

5.2 Names 199

5.3 Variables 200

5.4 The Concept of Binding 203

5.5 Scope 211

5.6 Scope and Lifetime 222

5.7 Referencing Environments 223

5.8 Named Constants 224

Summary • Review Questions • Problem Set • Programming Exercises 227

Module 1.2

Names, Bindings, and Scopes

5.1 Introduction 198

- Imperative languages are abstractions of von Neumann architecture
 - Memory: stores both instructions and data
 - Processor: provides operations for modifying the contents of memory •

Variables are characterized by a collection of properties or attributes

- The most important of which is **type**, a fundamental concept in programming languages
- To design a type, must consider scope, lifetime, type checking, initialization, and type compatibility

5.2 Names 199

5.2.1 Design issues

- The following are the primary design issues for names:
 - Maximum length?
 - Are names case sensitive?
 - Are special words reserved words or keywords?

5.2.2 Name Forms

- A **name** is a string of characters used to identify some entity in a program.
- Length
 - If too short, they cannot be connotative –
 - Language examples:
 - FORTRAN I: maximum 6 ▪
 - COBOL: maximum 30
 - C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31
 - C# and Java: **no limit**, and all characters are significant

- C++: **no limit**, but implementers often impose a length limitation because they do not want the **symbol table** in which identifiers are stored during compilation to be too large and also to simplify the maintenance of that table.
- Names in most programming languages have the same form: a letter followed by a string consisting of letters, digits, and (`_`). Although the use of the `_` was widely used in the 70s and 80s, that practice is far less popular.
- **C-based** languages (C, Objective-C, C++, Java, and C#), replaced the `_` by the “camel” notation, as in `myStack`.

- Prior to Fortran 90, the following two names are equivalent:

```
Sum Of Salaries SumOfSalaries    // names could
                                   have embedded
                                   spaces // which
                                   were ignored
```

- Special characters
 - PHP: all variable names must begin with dollar signs \$
 - Perl: all variable names begin with special characters \$, @, or %, which specify the variable's type
 - if a name begins with \$ is a scalar, if a name begins with @ it is an array, if it begins with %, it is a hash structure
 - Ruby: variable names that begin with @ are instance variables; those that begin with @@ are class variables
- Case sensitivity
 - Disadvantage: readability (names that look alike are different) – Names in the C-based languages are case sensitive
 - Worse in C++, Java, and C# because predefined names are mixed case (e.g. IndexOutOfBoundsException)
 - In C, however, exclusive use of lowercase for names.
 - C, C++, and Java names are case sensitive rose, Rose, ROSE are distinct names “What about Readability”

5.2.3 Special words

- An aid to readability; used to delimit or separate statement clauses
- A **keyword** is a word that is special only in certain contexts.
- Ex: Fortran

```
Real Apple                        // Real is a data type followed
Real = 3.4                        with a name, therefore Real is a
                                   keyword
                                   // Real is a variable name
```

- Disadvantage: poor readability. Compilers and users must recognize the difference. •
- A **reserved** word is a special word that **cannot** be used as a user-defined name.
- Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has **300** reserved words!)
- As a language design choice, reserved words are **better** than keywords.

- Ex: In Fortran, they are **only** keywords, which means they can be redefined. One could have the statements:

Integer Real

Real Integer

```
// keyword  
"Integer"  
and  
variable  
"Real" //  
keyword  
"Real" and  
variable  
"Integer"
```

5.3 Variables 200

- A variable is an abstraction of a memory cell.
- Variables can be characterized as a sextuple of attributes:
 - Name
 - Address –
 - Value
 - Type
 - Lifetime –
 - Scope
- Name
 - Not all variables have names: Anonymous, heap-dynamic variables •

Address

- The memory address with which it is associated
- A variable may have **different** addresses at **different** times during execution. If a subprogram has a local var that is allocated from the run time stack when the subprogram is called, different calls may result in that var having different addresses.
- The address of a variable is sometimes called its ***l-value*** because that is what is required when a variable appears in the **left** side of an assignment statement.
- Aliases
 - If two variable names can be used to access **the same** memory location, they are called aliases
 - Aliases are created via **pointers, reference variables**, C and C++ **unions**.
 - Aliases are harmful to readability (program readers must remember all of them) •

Type

- Determines the **range** of values of variables and the set of **operations** that are defined for values of that type; in the case of floating point, type also determines the precision.
- For example, the int type in Java specifies a value range of -2147483648 to 2147483647, and arithmetic operations for addition, subtraction, multiplication, division, and modulus.
- Value
 - The value of a variable is the **contents** of the memory cell or cells associated with the variable.
 - Abstract memory cell - the physical cell or collection of cells associated with a variable.
 - A variable's value is sometimes called its ***r-value*** because that is what is required when a variable appears in the **right** side of an assignment statement.
 - The ***l-value*** of a variable is its address. ▪
 - The ***r-value*** of a variable is its value.

5.4 The Concept of Binding 203

- A **binding** is an association, such as between an attribute and an entity, or between an operation and a symbol.
- **Binding time** is the time at which a binding takes place.
- Possible binding times:
 - Language design time: bind operator symbols to operations.
 - For example, the asterisk symbol (*) is bound to the multiplication operation. –
 - Language implementation time:
 - A data type such as **int** in C is bound to a **range** of possible values.
 - Compile time: bind a variable to a **particular data type** at compile time. –
 - Load time: bind a variable to a **memory cell** (ex. C **static** variables)
 - Runtime: bind a **nonstatic** local variable to a memory cell.

5.4.1 Binding of Attributes to Variables

- A binding is **static** if it first occurs **before** run time and remains unchanged throughout program execution.
- A binding is **dynamic** if it first occurs **during** execution or can change during execution of the program.

5.4.2 Type Bindings

5.4.2.1 Static Type Bindings

- If static, the type may be specified by either an **explicit** or an **implicit** declaration.
- An **explicit** declaration is a program statement used for declaring the types of variables.
- An **implicit** declaration is a **default** mechanism for specifying types of variables (the first appearance of the variable in the program.)
- Both explicit and implicit declarations create static bindings to types.
- Fortran, PL/I, Basic, and Perl provide implicit declarations.
- EX:
 - In **Fortran**, an identifier that appears in a program that is not explicitly declared is implicitly declared according to the following convention:
I, J, K, L, M, or N or their lowercase versions is **implicitly** declared to be Integer type; otherwise, it is implicitly declared as Real type.
 - Advantage: writability.

- Disadvantage: reliability suffers because they prevent the compilation process from detecting some typographical and programming errors.
- In Fortran, vars that are accidentally left undeclared are given default types and unexpected attributes, which could cause subtle errors that, are difficult to diagnose.
- Less trouble with **Perl**: Names that begin with \$ is a scalar, if a name begins with @ it is an array, if it begins with %, it is a hash structure.
- In this scenario, the names @apple and %apple are unrelated.

- **Type Inference:** Some languages use type inferencing to determine types of variables (context)
 - **C#** - a variable can be declared with **var** and an initial value. The initial value sets the type

```
var sum = 0;           // sum is int
var total = 0.0; var name = //
"Fred";              to
                     ta
                     l
                     is
                     fl
                     oa
                     t
                     //
                     na
                     me
                     is
                     st
                     ri
                     ng
```

- **Visual Basic, ML, Haskell, and F#** also use type inferencing. The context of the appearance of a variable determines its type

5.4.2.2 Dynamic Type Bindings

- With dynamic type binding, the type of a variable is not specified by a declaration statement, nor can it be determined by the spelling of its name. Instead, the variable is bound to a type when it is assigned a value in an **assignment** statement.
- Dynamic Type Binding: In **Python, Ruby, JavaScript, and PHP**, type binding is dynamic
- Specified through an assignment statement
- Ex, JavaScript

```
list = [2, 4.33, 6, 8];  single-dimensioned array
list = 47;              scalar variable
```

- Advantage: **flexibility** (generic program units)
- Disadvantages:
 - **High cost** (dynamic type checking and interpretation)
 - Dynamic type bindings must be implemented using pure interpreter **not** compilers.
 - Pure interpretation typically takes at least **10** times as long as to execute equivalent machine code.
 - Type error detection by the **compiler** is difficult because any variable can be assigned a value of any type.

- Incorrect types of right sides of assignments are not detected as errors; rather, the type of the left side is simply changed to the incorrect type.
- Ex, JavaScript

`i, x` Integer

`y` floating-point array

`i = x;` what the user meant to type

but because of a keying error, it has the assignment statement

`i = y;` what the user typed instead

- **No error** is detected by the compiler or run-time system. `i` is simply changed to a floating-point array type. Hence, the result is erroneous. In a static type binding language, the compiler would detect the error and the program would not get to execution.

5.4.3 Storage Bindings and Lifetime

- **Allocation** - getting a cell from some pool of available cells.
- **Deallocation** - putting a cell back into the pool.
- The **lifetime** of a variable is the time during which it is bound to a particular memory cell. So the lifetime of a var begins when it is bound to a specific cell and ends when it is unbound from that cell.
- Categories of variables by lifetimes:
 - **static**,
 - **stack-dynamic**,
 - **explicit heap-dynamic**, and –
implicit heap-dynamic

5.4.3.1 Static Variables

- Static variables are bound to memory cells **before** execution begins and remains bound to the same memory cell throughout execution
- e.g. all FORTRAN 77 variables, C **static variables** in functions
- Advantages:
 - **Efficiency** (direct addressing): All addressing of static vars can be direct. No run-time overhead is incurred for allocation and deallocation of static variables.
 - **History-sensitive**: have vars retain their values between separate executions of the subprogram.
- Disadvantage:
 - Storage **cannot** be shared among variables.
 - Ex: if two large arrays are used by two subprograms, which are never active at the same time, they cannot share the same storage for their arrays.

5.4.3.2 Stack-dynamic Variables

- Storage bindings are created for variables when their declaration statements are elaborated, but whose types are statically bound.
- Elaboration of such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached.
- The variable declarations that appear at the beginning of a **Java method** are elaborated when the method is invoked and the variables defined by those declarations are deallocated when the method completes its execution.
- Stack-dynamic variables are allocated from the **run-time stack**.
- If scalar, all attributes except address are statically bound.

- **Local variables** in C subprograms and Java methods. •

Advantages:

- Allows recursion: each active copy of the recursive subprogram has its own version of the local variables.
- In the absence of recursion, it conserves storage b/c all subprograms share the same memory space for their locals.

- Disadvantages:
 - Overhead of allocation and deallocation. –
 - Subprograms **cannot** be history sensitive.
 - Inefficient references (indirect addressing) is required b/c the place in the stack where a particular var will reside can only be determined during execution.
- In Java, C++, and C#, variables defined in **methods** are by **default** stack-dynamic.

5.4.3.3 Explicit Heap-dynamic Variables

- Nameless memory cells that are allocated and deallocated by explicit directives “run-time instructions”, specified by the programmer, which take effect during execution.
- These vars, which are allocated from and deallocated to the heap, can only be referenced through pointers or reference variables.
- The **heap** is a collection of storage cells whose organization is highly disorganized b/c of the unpredictability of its use.
- e.g. Dynamic objects in C++ (via **new** and **delete**)

```
int *intnode;      // create a pointer
. . .
intnode = new int; // allocates the heap-dynamic variable .
. . .
delete intnode;    // deallocates the heap-dynamic variable
                  // to which intnode points
```

- An explicit heap-dynamic variable of int type is created by the new operator. –
- This operator can be referenced through the pointer, intnode.
- The var is deallocated by the **delete** operator.

- In Java, all data except the primitive scalars are **objects**.
 - Java objects are explicitly heap-dynamic and are accessed through **reference variables**. –
 - Java uses **implicit garbage collection**.
- Explicit heap-dynamic vars are used for dynamic structures, such as linked lists and trees that need to grow and shrink during execution.
- Advantage:

- Provides for dynamic storage management. •

Disadvantage:

- Inefficient “Cost of allocation and deallocation” and unreliable “difficulty of using pointer and reference variables correctly”

5.4.3.4 Implicit Heap-dynamic Variables

- Bound to heap storage only when they are assigned value. Allocation and deallocation caused by **assignment** statements.
- All their attributes are bound every time they are assigned.
- e.g. all variables in APL; all strings and arrays in Perl and JavaScript, and PHP.
- Ex, JavaScript

```
highs = [74, 84, 86, 90, 71];  an array of 5 numeric values
```

- Advantage:
 - Flexibility allowing generic code to be written. •

Disadvantages:

- Inefficient, because all attributes are dynamic “run-time.” –
- Loss of error detection by the compiler.

5.5 Scope 211

- The scope of a variable is the range of statements in which the variable is visible.
- A variable is **visible** in a statement if it can be referenced in that statement.
- **Local variable** is local in a program unit or block if it is declared there.
- **Non-local variable** of a program unit or block are those that are visible within the program unit or block but are not declared there.

5.5.1 Static Scope

- ALGOL 60 introduced the method of binding names to non-local vars is called **static scoping**.
- Static scoping is named because the scope of a variable can be statically determined – that is **prior** to execution.
- This permits a human program reader (and a compiler) to determine the type of every variable in the program simply by examining its source code.
- There are two categories of static scoped languages:
 - Nested Subprograms.
 - Subprograms that cannot be nested.
- Ada, and JavaScript, Common Lisp, Scheme, F#, and Python allow **nested** subprograms, but the C-based languages do **not**.
- When a compiler for static-scoped language finds a reference to a var, the attributes of the var are determined by finding the statement in which it was declared.
- For example:
 - Suppose a reference is made to a var `x` in subprogram `sub1`. The correct declaration is found by first searching the declarations of subprogram `sub1`.
 - If no declaration is found for the var there, the search continues in the declarations of the subprogram that declared subprogram `sub1`, which is called its **static parent**.
 - If a declaration of `x` is not found there, the search continues to the next larger enclosing unit (the unit that declared `sub1`'s parent), and so forth, until a declaration for `x` is found or the largest unit's declarations have been searched without success. an undeclared var error has been detected.
 - The static parent of subprogram `sub1`, and its static parent, and so forth up to and including the main program, are called the static **ancestors** of `sub1`.
- Ex: JavaScript function, `big`, in which the two functions `sub1` and `sub2` are nested:

```
function big() {  
  function sub1() {  
    var x = 7;  
    sub2();  
  }  
  function sub2() {  
    var y = x;  
  }  
  var x = 3;  
  sub1();  
}
```

- Under static scoping, the reference to the variable `x` in `sub2` is to the `x` declared in the procedure `big`.
 - This is true because the search for `x` begins in the procedure in which the reference occurs, `sub2`, but no declaration for `x` is found there.
 - The search thus continues in the static parent of `sub2`, `big`, where the declaration of `x` is found.
 - The `x` declared in `sub1` is ignored, because it is **not** in the static ancestry of `sub2`. •

The variable `x` is declared in both `big` and `sub1`, which is nested inside `big`.

- Within `sub1`, every simple reference to `x` is to the local `x`. –
- The outer `x` is **hidden** from `sub1`

5.5.2 Blocks

- From ALGOL 60, allows a section of code to have its own local variables whose scope is minimized.
- Such variables are **stack dynamic**, so they have their storage allocated when the section is entered and deallocated when the section is exited.
- The **C-based** languages allow any compound statement (a statement sequence surrounded by matched braces) to have declarations and thereby defined a new scope.
- Ex: Skeletal C function:

```
void sub() {  
    int count;  
    . . .  
    while ( . . . ) {  
        int count;  
        count ++;  
        . . .  
    }  
    . . .  
}
```

- The reference to `count` in the while loop is to that loop's local `count`. The `count` of `sub` is **hidden** from the code inside the while loop.
- A declaration for a var effectively hides any declaration of a variable with the same name in a larger enclosing scope.
- Note that this code is **legal** in C and C++ but **illegal** in Java and C#
- Most functional languages (Scheme, ML, and F#) include some form of **let** construct
- A let construct has two parts
 - The first part binds names to values
 - The second part uses the names defined in the first part •

Ex. Scheme:

```
(LET (  
    (name1 expression1)  
    . . .  
    (namen expressionn)  
)
```

– Consider the following call to LET:

```
(LET (  
    (name1 expression1)  
    . . .  
    (namen expressionn)  
)
```

```
(top (+ a b))  
(bottom (- c d))  
(/ top bottom)  
)
```

- This call computes and returns the value of the expression $(a + b) / (c - d)$

5.5.3 Declaration Order

- C99, C++, Java, and C# allow variable declarations to appear **anywhere** a statement can appear
- In C99, C++, and Java, the scope of all local variables is **from** the declaration to the end of the block
- In C#, the scope of any variable declared in a block is the **whole** block, regardless of the position of the declaration in the block
- However, a variable still must be declared before it can be used
- For example, consider the following C# code:

```
{int x;
  . . .
  {int x; // Illegal .
    . . .
  }
  . . .
}
```

- Because the scope of a declaration is the whole block, the following nested declaration of x is also illegal:

```
{
  . . .
  {int x; // Illegal .
    . . .
  }
  int x;
}
```

- Note that C# still requires that all be declared before they are used
- In C++, Java, and C#, variables can be declared in for statements
 - The scope of such variables is restricted to the for construct

```
void fun() {
  . . .
  for (int count = 0; count < 10; count++) {
    int count;
    . . .
  }
  . . .
}
```

- The scope of count is from the for statement to the end of for its body (the right brace)

5.5.4 Global Scope

- C, C++, PHP, and Python support a program structure that consists of a sequence of function definitions in a file

- These languages allow variable declarations to appear outside function definitions •

For example, C and C++ have both declarations and definitions of global data

- A declaration outside a function definition specifies that it is defined in another file –
 - A global variable in C is implicitly visible in all subsequent functions in the file.
 - A global variable that is defined after a function can be made visible in the function by declaring it to be external, as the in the following:

```
extern int sum;
```

5.5.5 Evaluation of Static Scoping

- Works well in many situations
- Problems:
 - In most cases, it allows more access to both variables and subprograms that is necessary
 - As a program evolves, the initial structure is destroyed and local variables often become **global**; subprograms also gravitate toward become global, rather than nested
- An alternative to the use of static scoping to control access to variables and subprograms is an **encapsulation** construct.

5.5.6 Dynamic Scope

- The scope of variables in APL, SNOBOL4, and the early versions of LISP is dynamic. **Perl** and Common Lisp also allow variables to be declared to have dynamic scope, although the default scoping mechanism in these languages is static.
- Dynamic Scoping is based on **calling sequences** of program units, not their textual layout (temporal versus spatial) and thus the scope is determined only at **run time**.
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.
- Ex: Consider again the function `big` from Section 5.5.1, which the two functions `sub1` and `sub2` are nested:

```
function big() {  
    function sub1() {  
        var x = 7;  
        . . .  
    }  
    function sub2() {  
        var y = x;  
        . . .  
    }  
    var x = 3;  
    . . .  
}
```

- Consider the two different call sequences for `sub2`:
 - `big` calls `sub2` and `sub2` uses `x`
 - The dynamic parent of `sub2` is `big`. The reference is to the `x` in **`big`**. –
 - `big` calls `sub1`, `sub1` calls `sub2`, and `sub2` uses `x`
 - The search proceeds from the local procedure, `sub2`, to its caller, **`sub1`**, where a declaration of `x` is found.
 - Note that if **static scoping** was used, in either calling sequence the reference to `x` in `sub2` is to `big`'s `x`.

5.5.7 Evaluation of Static Scoping

- Advantage: convenience
- Disadvantages:
 - While a subprogram is executing, its variables are visible to **all** subprograms it calls – Inability to **type check** references to nonlocals statically
 - Difficult to read, because the calling sequence of subprograms must be known to determine the meaning of references to nonlocal variables

- Finally, accesses to nonlocal variables in dynamic-scoped languages take for **longer** than access to nonlocals when static scoping is used

5.6 Scope and Lifetime 222

- Scope and lifetime are sometimes closely related, but are different concepts
- For example, In a Java method
 - The scope of such a variable is from its **declaration** to the end of the method
 - The lifetime of that variable is the period of **time** beginning when the method is entered and ending when execution of the method terminates
- Consider a **static** variable in a C or C++ function
 - Statically bound to the scope of that function and is also statically bound to storage
 - Its scope is static and local to the function but its lifetime extends over the **entire** execution of the program of which it is a part
- Ex: C++ functions

```
void printhead() {  
    . . .  
} /* end of printhead */  
void compute() {  
    int sum;  
    . . .  
    printhead();  
} /* end of compute */
```

- The **scope** of `sum` is contained within `compute` function
- The **lifetime** of `sum` extends over the time during which `printhead` executes.
- Whatever storage location `sum` is bound to before the call to `printhead`, that binding will continue during and after the execution of `printhead`.

5.7 Referencing Environments 223

- The referencing environment of a statement is the **collection** of all names that are **visible** in the statement
- In a **static-scoped** language, it is the local variables plus all of the visible variables in all of the enclosing scopes
- The referencing environment of a statement is needed while that statement is being compiled, so code and data structures can be created to allow references to variables from other scopes during run time.
- A subprogram is **active** if its execution has begun but has not yet terminated.
- In a **dynamic-scoped** language, the referencing environment is the local variables plus all visible variables in all active subprograms.
- Ex, Python skeletal, **static-scoped language**

```

g = 3;                                     # A global

def sub1(): a = 5;
    b = 6; . . .
    def sub2(): global g; c =
        9;
        . . .
        def sub3(): nonlocal c; g =
            11;
            . . .
# Crates a local
# Crates
another
local 1
# Global g is now
assignable here #
Creates a new local
2
# Makes nonlocal c
visible here #
Creates a new local
3

```

- The referencing environments of the indicated program points are as follows:

Point	Referencing Environment
1	local a and b (of sub1), global g for reference, but not for assignment local c (of sub2), global g for both reference and for assignment Note: a and b (of sub1) for reference, but not for assignment
3	nonlocal c (of sub2), local g (of sub3) Note: a and b (of sub1) for reference, but not for assignment

- Ex, **Dynamic-scoped language**
- Consider the following program; assume that the only function calls are the following: `main` calls `sub2`, which calls `sub1`

```

void sub1( ) {
    int a, b;

    . . .                               1 }
    /* end of sub1 */ void
sub2( ) {
    int b, c;

    . . .                               2
    sub1();
}      /* end of sub2 */
void main( ) {
    int c, d;

    . . .                               3
    sub2( );
}      /* end of main */

```

- The referencing environments of the indicated program points are as follows:

Point	Referencing Environment
1	a and b of sub1, c of sub2, d of main (c of main, b of sub2 hidden) 2
2	b and c of sub2, d of main (c of main is hidden)
3	c and d of main

5.8 Named Constants 224

- It is a variable that is bound to a value only at the time it is bound to storage; its value **cannot** be change by assignment or by an input statement.
- Ex, Java

```
final int LEN = 100;
```

- Advantages: readability and modifiability

Variable Initialization

- The binding of a variable to a value at the time it is bound to storage is called initialization.
- Initialization is often done on the declaration statement.
- Ex, C++

```
int sum = 0;  
int* ptrSum = &sum;  
char name[] = "George Washington Carver";
```


Summary

- Variables are characterized by the 6 of attributes:
 - Name
 - Address – Value
 - Type
 - Lifetime – Scope
- Binding is the association of attributes with program entities. Binding can be static or dynamic type binding.
 - **Static type binding:**
 - A binding is **static** if it first occurs **before** run time and remains unchanged throughout program execution.
 - Declaration either explicit or implicit, provide a means of specifying the static binding of variables to types
 - **Dynamic type binding:**
 - A binding is **dynamic** if it first occurs **during** execution or can change during execution of the program.
 - It allows greater flexibility but at the expense of readability, efficiency, and reliability
- Scalar variables can be separated into 4 categories:
 - **Static Variables**
 - **Stack Dynamic Variables**
 - **Explicit Heap Dynamic Variables – Implicit Heap Dynamic Variables**
- The scope of a variable is the range of statements in which the variable is visible.
 - **Static scope:**
 - Static scoping is named because the scope of a variable can be **statically** determined – that is **prior** to execution
 - This permits a human program reader (and a compiler) to determine the type of every variable in the program simply by examining its source code.
 - It provides a simple, reliable, and efficient method of allowing visibility of nonlocal variables in subprograms
 - **Dynamic scope:**
 - It is based on **calling sequences** of program units, not their textual layout and thus the scope is determined only at **run time**.

- It provides more flexibility than static scoping but, again, at expense of readability, reliability, and efficiency