

## Chapter 4 Names

*The first step toward wisdom is calling things by their right names.  
Anon. Chinese Proverb*

- 4.1 Syntactic Issues
- 4.2 Variables
- 4.3 Scope
- 4.4 Symbol Table
- 4.5 Resolving References
- 4.6 Dynamic Scoping
- 4.7 Visibility
- 4.8 Overloading
- 4.9 Lifetime

### Bindings

- A binding is an association between an entity (such as a variable) and a property (such as its value).
  - A binding is static if the association occurs before run-time.
  - A binding is dynamic if the association occurs at run-time.
- Static binding is used by most major languages
- Some other languages delay name resolution until runtime

### Visibility and Lifetime

- We also need to consider *visibility* (scope) and *lifetime* of names
  - The lifetime of a variable name refers to the time interval during which memory is allocated.
  - The scope or visibility of a variable refers to the parts of the program where the variable is accessible
- While lifetime is closely related to visibility (variables local to a function typically exist only for the lifetime of the function) it is not the same as visibility

### Two major language classes

- We will consider a broad division of imperative and imperative/OO languages in discussing names because the treatment of names falls broadly into two classes:
  - C-Like: C, C++, Java, etc (AKA the curly-brace languages)
  - Pascal-Like: Pascal, Ada, Modula, etc.

### Syntactic Issues

- Names or identifiers denote many classes of things:
  - Variables, types, functions, classes, modules ...
- Lexical rules determine how a name should be constructed
  - Common example: identifiers must start with a letter or underscore, followed by letters, digits or underscores
- Many languages have “reserved words” that follow lexical rules for identifiers but may not be used as identifiers

## Case Sensitivity and Length of Names

- Case Sensitivity
  - *C-like: yes*
  - *Pascal-like: no*
  - *Early languages: no*
  - *PHP: partly yes, partly no*
- Early languages placed severe restrictions on length of names (e.g., Fortran 6 characters, some BASICs 2 characters)
  - *Derived from memory or architectural constraints*
- Nearly all languages place some restriction on name length
  - *Derived from internal structure of compiler's symbol table*

## Special Characters

- Many languages allow characters other than A-Z, a-z, and 0-9 in names
- Typically very few characters with syntactic restrictions
  - *Cobol allows hyphens in names (what sort of parsing problem does this present?)*
  - *C allows unrestricted use of the underscore*
- In some languages special characters convey syntactic or semantic meaning
  - *Perl \$foo is a scalar, @foo is an array*
  - *BASIC has type declaration chars: foo\$ is a string, foo! is a single precision float*

## Implicit Typing

- Found in FORTRAN 77 and some old BASICs
- In FORTRAN 77 variables beginning with I,J,K,L, M are implicitly integers, otherwise REAL is assumed
  - *COUNT is a real*
  - *KOUNT is an integer*
  - *IMPLICIT statement allows mods to this rule*
- QuickBasic has DEFINT, DEFSNG, etc. statements to define implicit typing

## Reserved Words

- Cannot be used as *Identifiers*
- Usually identify major constructs: *if while switch*
- Predefined identifiers (e.g., library routines) differ from reserved words – in some languages they can be redefined by the programmer
  - *Pascal: const true = false*
  - *Cobol: no redefinition allowed; there are hundreds of reserved words and identifiers*
- PL/I had no reserved words!
  - *If if = then then else = if else if = else*

## 4.2 Variables

- Defn: A variable is a binding of a name to a *memory address*
  - *Addresses can refer to individual or aggregated memory locations – or even persistent storage*
- In addition to a name variables have a type, a value and a lifetime
- Any of these four bindings can be static or dynamic
- Binding rules have a significant bearing on language implementation

## L-Values and R-Values

- A distinction in the use of variable names first introduced in Algol 68
- L-value - use of a variable name to denote its address.
- R-value - use of a variable name to denote its value.
  - Ex:  $x = x + 1$*
  - *Store into memory at address of x the value of x plus one.*
- On the LHS x denotes an address while on the RHS it denotes a value

## Explicit Dereferencing and Pointers

- Some languages support/require explicit dereferencing.
- ML: `x := !y + 1`

- C pointers

```
int x,y;  
int *p;  
x = *p;  
*p = y;
```

- Note that C is liberal about whitespace:

```
x *y; /* y is a pointer to type x */  
x * y; /* same as above */
```

## Scope

- Defn: The *scope* of a name is the collection of statements which can access the name binding.
- Defn: In *static* scoping, a name is bound to a collection of statements according to its position in the source program.
- Also called *lexical* scoping – based on grammatical structure of the program
- Static scoping is performed at compile time and does not vary with the execution history of a program
- Used by most modern languages

## Nested and Disjoint scopes

- Name collision becomes increasingly important as programs and memories become larger
  - *Names with limited scope can be reused elsewhere*
- Two different scopes are either nested or disjoint.
- In disjoint scopes, same name can be bound to different entities without interference.
- Nested scopes are like nested loops: one scope is contained within another
- What constitutes a scope?

## Lexical units of scope

	Algol	C	Java	Ada
Package	n/a	n/a	yes	yes
Class	n/a	n/a	nested	yes
Function	nested	yes	yes	nested
Block	nested	nested	nested	nested
For Loop	no	no	yes	automatic

- This table is not definitive
- Note that the compilation unit always constitutes a scope in addition to the above units

## Lexical Units of Scope

- In Fortran 60 the only unit of scope was the compilation unit and all scopes were disjoint
- Algol 60 introduced nested scoping, including nested functions and a begin-end block of code that could include declarations and functions
- C and C++ do not allow nested functions but do allow nested blocks

## Defining Scope

- Defn: The scope in which a name is defined or declared is called its *defining scope*.
- Defn: A reference to a name is *nonlocal* if it occurs in a nested scope of the defining scope; otherwise, it is *local*.

## Nested Functions (Pascal Syntax)

```
function Foo(i: integer): integer
  function Bar(x: integer): integer
  begin
    bar := i * x
  end
begin
  Foo := Bar(42)
end
```

## Nested Functions

- Considered a useful tool for encapsulation and isolation of concerns by limiting visibility
  - *The OO paradigm provided a different mechanism for writing functions with limited visibility*
- Introduced with Algol; found in descendants such as Simula, Pascal, Modula2, Ada
- Found in many modern functional languages (Scheme, ML, Haskell)
- Not present in C, C++, Java
- Varying levels of support in Javascript, Actionscript, Perl, Ruby, Python

## C Example of Block Scope

```
1 void sort (float a[ ], int size) {
2   int i, j;
3   for (i = 0; i < size; i++) // i, size local
4     for (j = i + 1; j < size; j++)
5       if (a[j] < a[i]) { // a, i, j local
6         float t;
7         t = a[i]; // t local; a, i nonlocal
8         a[i] = a[j];
9         a[j] = t;
10      }
11 }
```

## Forward References

- A forward reference is a reference to variable (or other name) lexically before it is declared
- Many languages forbid forward references to simplify compiler design
- C specifies that all declarations must precede all other statements in a scope
- C++ and Java allow declarations anywhere, but forbid forward references
- To consider: which rule (C or C++) is more conducive to well organized code?

## For Loop Scope

- The FOR loop structure is found in many languages
- The loop control variable is often needed only for the scope of the for loop
- C++/Java/vb.net allow declaration in the loop statement:

```
for (int i = 0; i < 10; i++) {
  System.out.println(i);
  ...
}
```

- Ada automatically declares and limits the scope of a loop control variable

## Classes and Forward References

- Java and other languages such as C# and vb.net do not use a forward reference rule in class definitions
- Instance variables and methods can be declared and defined in any convenient order

## Symbol Table

- A *symbol table* is a data structure kept by a translator that allows it to keep track of each declared name and its binding.
- Assume for now that each name is unique within its local scope.
- The data structure can be any implementation of a dictionary, where the name is the key and a value (which can be a complex object) is associated with the name.

– Note: associative arrays in Perl, Javascript, PHP and other languages provide the same functionality

## Semantic Analysis

- For languages with static scoping the symbol table is constructed during semantic analysis and persists for the remainder of compilation
- Symbol tables are often saved in files for use by symbolic debuggers

## Algorithm for Handling Non-local References

1. Each time a scope is entered, push a new dictionary onto the stack.
2. Each time a scope is exited, pop a dictionary off the top of the stack.
3. For each name declared, generate an appropriate binding and enter the name-binding pair into the dictionary on the top of the stack.
4. Given a name reference, search the dictionary on top of the stack:
  - a) If found, return the binding.
  - b) Otherwise, repeat the process on the next dictionary down in the stack.
  - c) If the name is not found in any dictionary, report an error.

## Dictionaries for Fig 4.1

- Value associated with variable name is just a line number
- C program in Fig. 4.1, stack of dictionaries at line 7:

```
<t, 6>
<j, 4> <i, 3> <size,1> <a, 1>
<sort, 1>
```

- At line 4 and 11:

```
<j, 4> <i, 3> <size,1> <a, 1>
<sort, 1>
```

## Resolving References

- Defn: For static scoping, the *referencing environment* for a name is its defining scope and all nested subsopes.
- The referencing environment defines the set of statements which can validly reference a name.

## Fig 4.2

Shows nested and disjoint scopes:

Outer scope: vars on line 1 plus functions B, A, main

B's scope is param w + line 3

A's scope is params on line 8 plus vars on line 9

main scope is vars on line 15

```
1 int h, i;
2 void B(int w) {
3     int j, k;
4     i = 2*w;
5     w = w+1;
6     ...
7 }
8 void A (int x, int y) {
9     float i, j;
10    B(h);
11    i = 3;
12    ...
13 }
14 void main() {
15    int a, b;
16    h = 5; a = 3; b = 2;
17    A(a, b);
18    B(h);
19    ...
20 }
```

### Symbol tables for Fig 4.2

1. Outer scope: <h, 1> <i, 1> <B, 2> <A, 8><main, 14>
2. Function B: <w, 2> <j, 3> <k, 4>
3. Function A: <x, 8> <y, 8> <i, 9> <j, 9>
4. Function main: <a, 15> <b, 15>

### Symbol Table Stacks

Symbol Table Stack for Function B:

```
<w, 2> <j, 3> <k, 4>
<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
```

Symbol Table Stack for Function A:

```
<x, 8> <y, 8> <i, 9> <j, 9>
<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
```

Symbol Table Stack for Function main:

```
<a, 15> <b, 15>
<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
```

### Resolving Non-Local References

Line	Reference	Declaration
4	i	1
10	h	1
11	i	9
16	h	1
18	h	1

### Dynamic Scoping

- Defn: In *dynamic scoping*, a name is bound to its most recent declaration based on the program's call history.
- Used by early Lisp, APL, Snobol, Perl (which also supports static scoping)
- Symbol table for each scope built at compile time, but names are resolved at run time.
- Scope pushed/popped on stack when entered/exited.

### Fig 4.2

Call history:

```
main (17) → A (10) → B
main (18) → B
```

**Fn Dictionary**

```
B <w, 2> <j, 3> <k, 3>
A <x, 8> <y, 8> <i, 9> <j, 9>
main <a, 15> <b, 15>
    <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
```

Reference to i(4) resolves to <i, 9> in A.

```
B <w, 2> <j, 3> <k, 3>
main <a, 15> <b, 15>
    <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
```

Reference to i(4) resolves to <i, 1> in global scope.

```
1 int h, i;
2 void B(int w) {
3     int j, k;
4     i = 2*w;
5     w = w+1;
6     ...
7 }
8 void A (int x, int y) {
9     float i, j;
10    B(h);
11    i = 3;
12    ...
13 }
14 void main() {
15    int a, b;
16    h = 5; a = 3; b = 2;
17    A(a, b);
18    B(h);
19    ...
20 }
```

### Problems of Dynamic Scoping

- Name resolution requires knowledge of runtime history; different runtime history can lead to different scope resolution for variables
  - Tends to be very error prone and leads to difficult to understand programs
  - Pretty much abandoned in modern languages except for Common Lisp and Perl
    - Variables declared with "my" in Perl have static scope; declared with "local" have dynamic
- ```
my $x;
local $y;
```

## Advantages of Static Scoping

- Static type-checking of refs to non-local vars
- Local variables are NOT visible to called function
- Dynamic scoping requires maintenance and traversal of binding stacks; static references are directly translated to memory accesses

## Visibility

- Defn: A name is *visible* if its referencing environment includes the reference and the name is not redeclared in an inner scope.
- A name redeclared in an inner scope effectively *hides* the outer declaration.
- Some languages provide a mechanism for referencing a hidden name; e.g.: `this.x` in C++/Java.

```
Public Class Student {
    private String name;
    public Student (String name, ...) {
        this.name = name;
    }
}
```

## Ada Example

```
procedure Main is
  x : Integer;
  procedure p1 is
    x : Float;
    procedure p2 is
      begin
        ... x ...
      end p2;
    begin
      ... x ...
    end p1;
  procedure p3 is
    begin
      ... x ...
    end p3;
begin
  ... x ...
end Main;
```

References to x in Main and p3 refer to the integer x declared in Main

References in p1 and p2 refer to the float x declared in p1

But we could use explicit main.x in p1 or p2 to refer to the integer variable

## Overloading

- Defn: *Overloading* uses the number or type of parameters to distinguish among identical function names or operators.

Examples:

- +, -, \*, / can be float (32,64,80 bits) or int (8,16,32,64 bits)
- + can be float or int addition or string concatenation in Java, SQL, Javascript, VB etc.
- `System.out.print(x)` in Java

## Modula and Pascal I/O Statements

- Historically overloading did not apply to user-defined functions and operators

Pascal: I/O statements specified in language

- `Read()` for all simple types

Modula: library functions

- `Read()` for characters
- `ReadReal()` for floating point
- `ReadInt()` for integers
- `ReadString()` for strings

## Ada Overloading

- Ada was the first modern programming to generalize the concept of overloading to include operators applied to programmer-defined types and functions

```
function "+"(In_Array1, In_Array2 : ARY_INT) return ARY_INT is
  Temp_Array : ARY_INT;
begin
  for Index in ARY_INT'RANGE loop
    Temp_Array(Index) := In_Array1(Index) + In_Array2(Index);
  end loop;
  return Temp_Array;
end "+";

function "mod"(In_Array1, In_Array2 : ARY_INT) return ARY_INT
is
  Temp_Array : ARY_INT;
begin
  for Index in ARY_INT'RANGE loop
    Temp_Array(Index) := In_Array1(Index) mod
  end loop;
  return Temp_Array;
end "mod";
```

## Other overloading examples

- C++ allows operators and methods to be overloaded
  - What does `a << 2;` mean?
- Java and many other languages (e.g., VB, C#) allow overloaded methods as long as the signatures (parameter count and types, return type) are unique

## Java Method Overloading

```
public class PrintStream extends
    FilterOutputStream {
    ...
    public void print(boolean b);
    public void print(char c);
    public void print(int i);
    public void print(long l);
    public void print(float f);
    public void print(double d);
    public void print(char[ ] s);
    public void print(String s);
    public void print(Object obj);
}
```

## Instance Variables and Methods

- Many OO languages allow an instance variable and a method to share a name because syntax of method invocation is different from a variable reference

```
Public Class Student {
    private String name;
    . . .
    public String name() { return name; }
    . . .
}
```

## Lifetime

- Defn: The *lifetime* of a variable is the time interval during which the variable has been allocated a block of memory.
- Early languages Fortran and Cobol used static (compile time) allocation.
  - Memory was allocated in a global memory area
  - Use of static global memory for function parameters and return addresses means that recursive functions cannot exist
  - All variables existed for the duration of the program
- Memory management was the programmer's responsibility
  - Early machines had very limited memory space e.g., IBM 1130 32KB; IBM 360 64KB

## Algol 60

- Algol introduced the notion that memory should be allocated/deallocated at scope entry/exit.
- This allows recursive functions to exist because the local memory for the function is allocated when the function is invoked
- Almost all languages use a stack for function local memory
  - Structure is often called a "stack frame"
  - Contains parameters, return addresses, local or automatic variables, pointers to stack frames for caller and/or outer scope
- Use of stack by HLLs led to development of stack based machines used in most modern architectures

## When scope != lifetime

- With static allocation a variable never "forgets" its value, even variables declared within a function scope
- With dynamic allocation variables are created and destroyed as the program runs.
  - Memory allocated on function entry is returned on exit and will be overwritten
- Most languages provide mechanisms that can be used to break the *scope equals lifetime* rule.

## Counting Function Invocations

- It is sometimes handy to know how often a particular function has been called:

```
Double func() {  
    count++  
    . . .  
}
```

- But if count is declared inside func it will be recreated with every invocation

## Global and Static Variables

- Variables declared outside of any function in C / C++ are global variables and are statically allocated for the life of the program (ex. Variable h in Fig 4.2)

- But we can also do this:

```
Double func() {  
    static int count = 0;  
    count++  
    . . .  
}
```

- Note that count is initialized to 0 during compilation (not runtime) and is never reinitialized

## Global and Static Variables in Java

- Java supports static variables
- But does not allow declaration of variables outside of any scope, so C/C++ style globals cannot be used
- Solution is to use public static variables in a class

```
public class GlobalData {  
    public static int usercount = 0;  
    public static long hitcount = 0;  
}
```

## Global Variables and Multiple Compilation Units

- In most languages that support global declarations are made at the compilation unit level (typically a single file)
- Additional syntactic mechanisms are needed to force the scope of a name to be global across compilation units
- C mechanisms are: header files and the include directive:

```
#include <foo.h>  
#include "bar.h"
```

- Or Extern declarations (resolved by linker)

```
extern int x;    /* in file 1 */  
int x;          /* in file 2 */
```