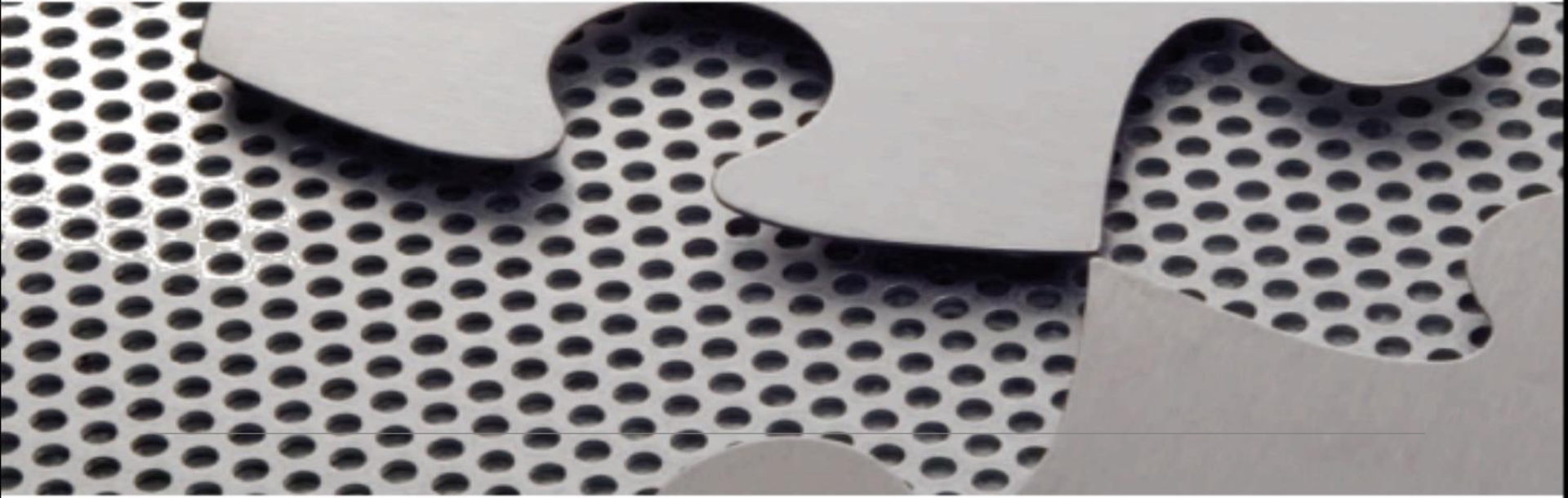


Languages Third Edition



CHAPTER 7

BASIC SEMANTICS

Objectives

Understand attributes, binding, and semantic functions

Understand declarations, blocks, and scope

Learn how to construct a symbol table

Understand name resolution and overloading

Understand allocation, lifetimes, and the environment

Work with variables and constants

Learn how to handle aliases, dangling references, and garbage

Introduction

Syntax: what the language constructs look like

Semantics: what the language constructs actually do

Specifying semantics is more difficult than specifying syntax

Several ways to specify semantics:

- Language reference manual
- Defining a translator
- Formal definition

Introduction

Language reference manual:

- Most common way to specify semantics
- Provides clearer and more precise reference manuals
- Suffers from a lack of precision inherent in natural language descriptions
- May have omissions and ambiguities

Introduction

Defining a **translator**:

- Questions about a language can be answered by experimentation
- Questions about program behavior cannot be answered in advance
- Bugs and machine dependencies in the translator may become part of the language semantics, possibly unintentionally
- May not be portable to all machines
- May not be generally available

Introduction

Formal definition:

- **Formal mathematical methods**: precise, but are also complex and abstract
- Requires study to understand
- **Denotational semantics**: probably the best formal method for the description of the translation and execution of programs
 - Describes semantics using a series of functions

Attributes, Binding, and Semantic Functions

Names (or **identifiers**): a fundamental abstraction mechanism used to denote language entities or constructs

Fundamental step in describing semantics is to describe naming conventions for identifiers

Most languages also include concepts of location and value

- **Value**: any storable quantities
- **Location**: place where value can be stored; usually a relative location

Attributes, Binding, and Semantic Functions

Attributes: properties that determine the meaning of the name to which they are associated

Example in C: `const int n = 5;`

- Attributes for variables and constants include data type and value

Example in C:

```
double f(int n) {  
    ...  
}
```

- Attributes include “function,” number, names and data type of parameters, return value data type, body of code to be executed

Attributes, Binding, and Semantic Functions

Assignment statements associate attributes to names

Example: `x = 2;`

- Associates attribute “value 2” to variable `x`

Example in C++:

```
int* y;  
y = new int;
```

- Allocates memory (associates location to `y`)
- Associates value

Attributes, Binding, and Semantic Functions

Binding: process of associating an attribute with a name

Binding time: the time when an attribute is computed and bound to a name

Two categories of binding:

- **Static binding:** occurs prior to execution
- **Dynamic binding:** occurs during execution

Static attribute: an attribute that is bound statically

Dynamic attribute: an attribute that is bound dynamically

Attributes, Binding, and Semantic Functions

Languages differ substantially in which attributes are bound statically or dynamically

- Functional languages tend to have more dynamic binding than imperative languages

Static attributes can be bound during translation, during linking, or during loading of the program

Dynamic attributes can be bound at different times during execution, such as entry or exit from a procedure or from the program

Attributes, Binding, and Semantic Functions

Some attributes are bound prior to translation time

- Predefined identifiers: specified by the language definition
- Values true/false bound to data type Boolean
- `maxint` specified by language definition and implementation

All binding times except execution time are static binding

Attributes, Binding, and Semantic Functions

A translator creates a data structure to maintain bindings

- Can be thought of as a function that expresses the binding of attributes to names

Symbol table: a function from names to attributes

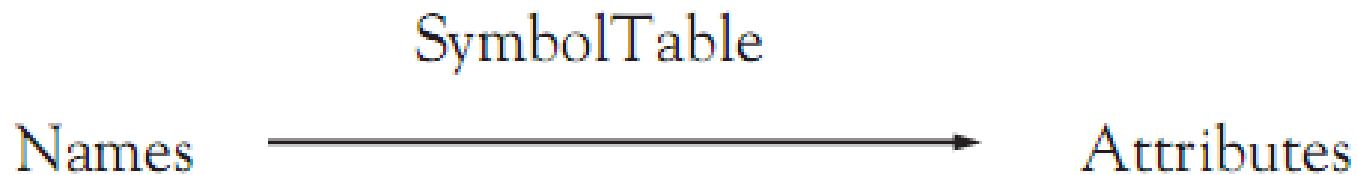


Figure 7.1 Mapping names to attributes in a symbol table

Attributes, Binding, and Semantic Functions

Parsing phase of translation includes three types of analysis:

- **Lexical analysis**: determines whether a string of characters represents a token
- **Syntax analysis**: determines whether a sequence of tokens represents a phrase in the context-free grammar
- **Static semantic analysis**: establishes attributes of names in declarations and ensures that the use of these names conforms to their declared attributes

During execution, attributes are also maintained

Attributes, Binding, and Semantic Functions



Figure 7.2 Mapping names to locations in an environment

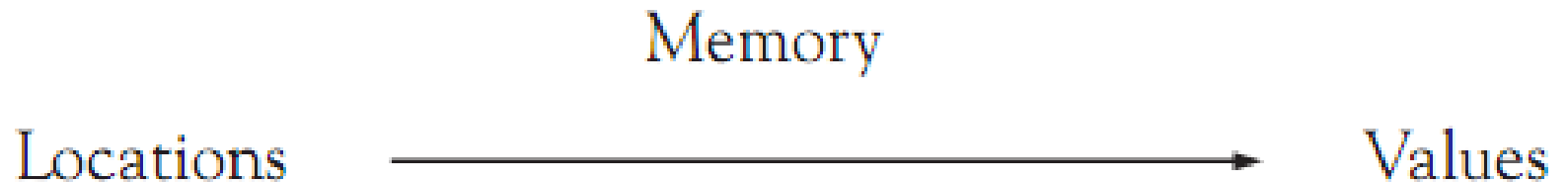


Figure 7.3 Mapping locations to values in a memory

Declarations, Blocks, and Scope

Bindings can be **implicit** or **explicit**

Example: `int x;`

- Data type is bound explicitly; location of `x` is bound implicitly

Entire declaration itself may be implicit in languages where simply using the variable name causes it to be declared

Definition: in C and C++, a declaration that binds all potential attributes

Prototype: function declaration that specifies the data type but not the code to implement it

Declarations, Blocks, and Scope

Block: a sequence of declarations followed by a sequence of statements

Compound statements: blocks in C that appear as the body of functions or anywhere an ordinary program statement could appear

Local declarations: associated with a block

Nonlocal declarations: associated with surrounding blocks

Block-structured languages allow nesting of blocks and redeclaration of names within nested blocks

Declarations, Blocks, and Scope

Each declared name has a **lexical address** containing a **level number** and an **offset**

- Level number starts at 0 and increases into each nested block

Other sources of declarations include:

- A `struct` definition composed of local (member) declarations
- A class in object-oriented languages

Declarations can be collected into packages (Ada), modules (ML, Haskell, Python), and namespaces (C++)

Declarations, Blocks, and Scope

Scope of a binding: region of the program over which the binding is maintained

Lexical scope: in block-structured languages, scope is limited to the block in which its associated declaration appears (and other blocks contained within it)

Declaration before use rule: in C, scope of a declaration extends from the point of declaration to the end of the block in which it is located

```
(1)  int x;

(2)  void p() {
(3)      char y;
(4)      ...
(5)  } /* p */

(6)  void q() {
(7)      double z;
(8)      ...
(9)  } /* q */

(10) main() {
(11)     int w[10];
(12)     ...
(13) }
```

Figure 7.4 Simple C program demonstrating scope

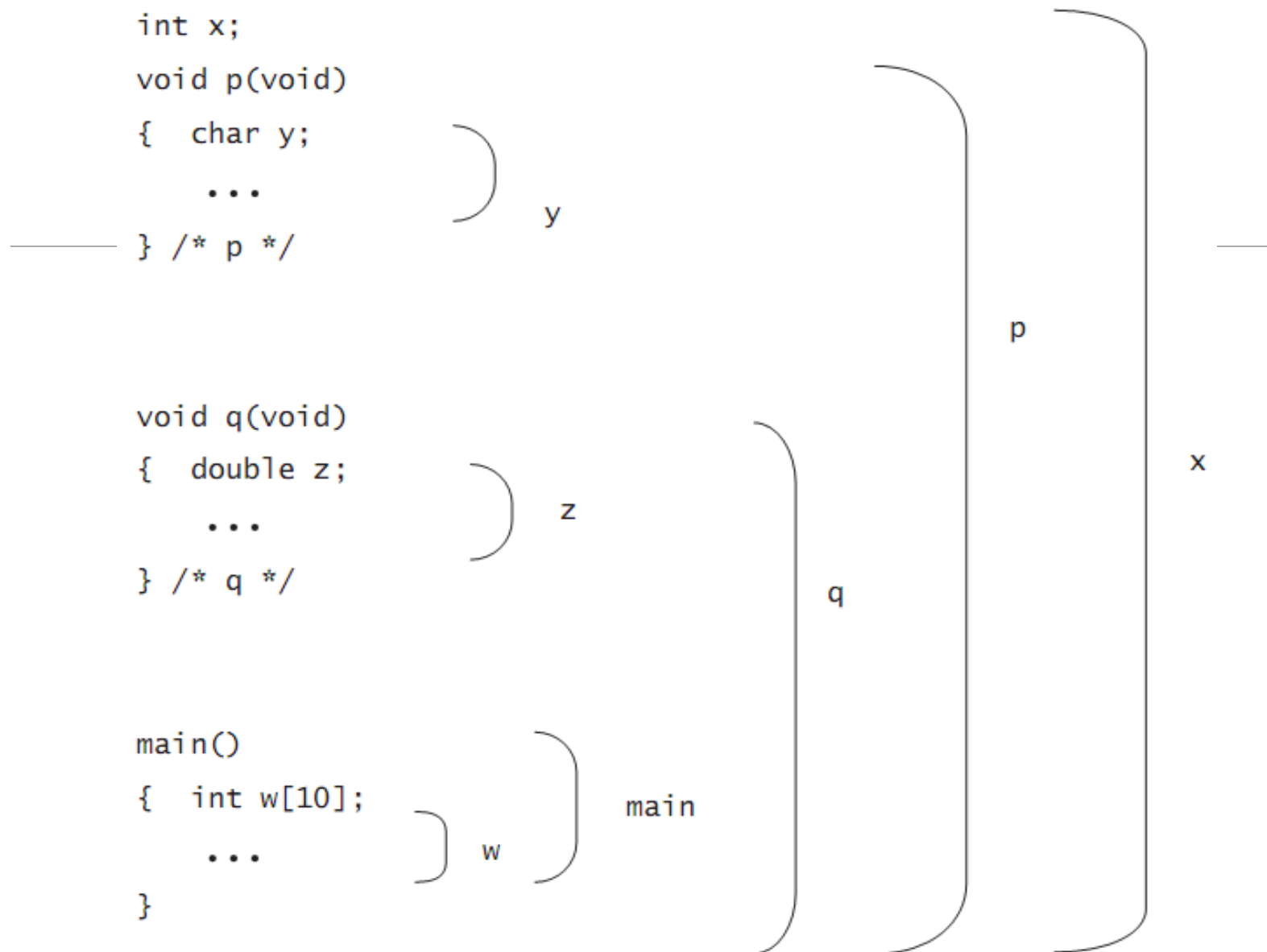


Figure 7.5 C Program from Figure 7.4 with brackets indicating scope

Declarations, Blocks, and Scope

Declarations in nested blocks take precedence over previous declarations

A global variable is said to have a **scope hole** in a block containing a local declaration with the same name

- Use **scope resolution operator** `::` in C++ to access the global variable

Local declaration is said to **shadow** its global declaration


Visibility: includes only regions where the bindings of a declaration apply

Declarations, Blocks, and Scope

```
int x;

void p(){
  char x;
  x = 'a'; // assigns to char x
  ::x = 42; // assigns to global int x
  ...
}

main(){
  x = 2; // assigns to global x
  ...
}
```



Declarations, Blocks, and Scope

Scope rules need to be constructed such that recursive (self-referential) declarations are possible when they make sense

- Example: functions must be allowed to be recursive, so function name must have scope beginning before the block of the function body

```
int factorial (int n){  
    /* scope of factorial begins here */  
    /* factorial can be called here */  
    ...  
}
```


The Symbol Table

Symbol table:

- Must support insertion, lookup, and deletion of names with associated attributes, representing bindings in declarations

A lexically scoped, block-structured language requires a stack-like data structure to perform **scope analysis**:

- On block entry, all declarations of that block are processed and bindings added to symbol table
- On block exit, bindings are removed, restoring any previous bindings that may have existed

```
(1) int x;  
(2) char y;  
  
(3) void p(){  
(4)     double x;  
(5)     ...  
(6)     { int y[10];  
(7)         ...  
(8)     }  
(9)     ...  
(10) }  
  
(11) void q(){  
(12)     int y;
```

```
(13)     ...  
(14) }  
  
(15) main(){  
(16)     char x;  
(17)     ...  
(18) }
```

The Symbol Table

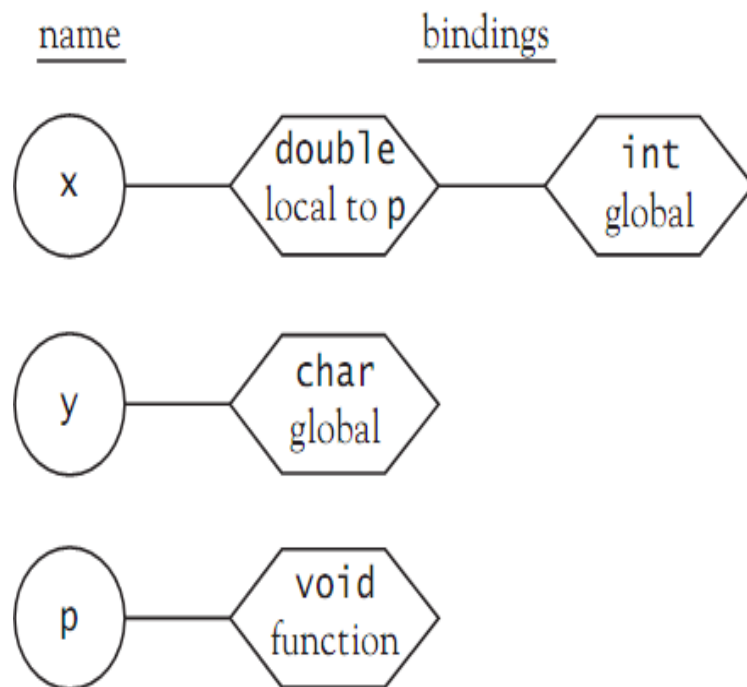


Figure 7.7 Symbol table structure at line 5 of Figure 7.6

```
(1) int x;  
(2) char y;  
  
(3) void p(){  
(4)     double x;  
(5)     ...  
(6)     { int y[10];  
(7)         ...  
(8)     }  
(9)     ...  
(10) }
```

```
(11) void q(){  
(12)     int y;
```

```
(13)     ...  
(14) }  
  
(15) main(){  
(16)     char x;  
(17)     ...  
(18) }
```

The Symbol Table

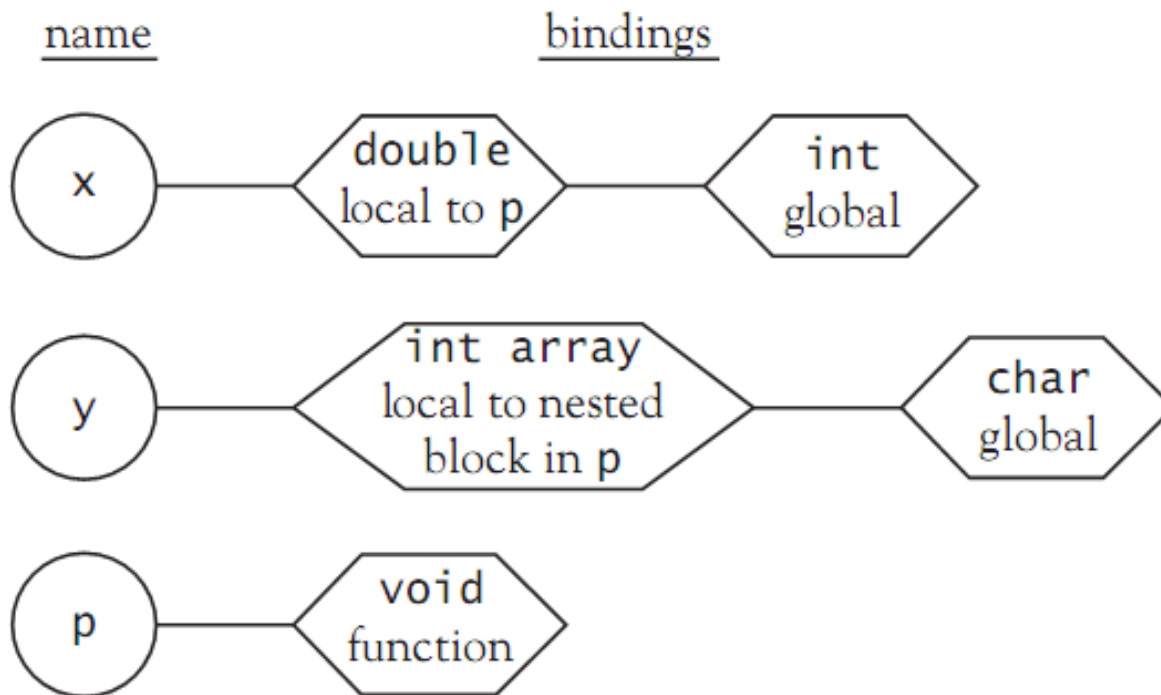


Figure 7.8 Symbol table structure at line 7 of Figure 7.6

```
(1) int x;  
(2) char y;  
  
(3) void p() {  
(4)     double x;  
(5)     ...  
(6)     { int y[10];  
(7)         ...  
(8)     }  
(9)     ...  
(10) }
```

```
(11) void q() {  
(12)     int y;
```

```
(13)     ...  
(14) }
```

```
(15) main() {  
(16)     char x;  
(17)     ...  
(18) }
```

The Symbol Table

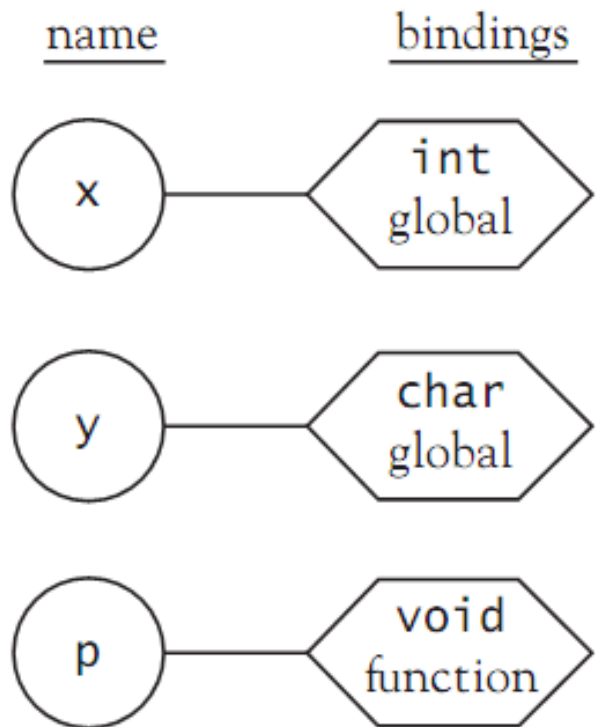


Figure 7.9 Symbol table structure at line 10 of Figure 7.6

```
(1) int x;  
(2) char y;  
  
(3) void p() {  
(4)     double x;  
(5)     ...  
(6)     { int y[10];  
(7)         ...  
(8)     }  
(9)     ...  
(10) }
```

```
(11) void q() {  
(12)     int y;
```

```
(13)     ...  
(14) }  
  
(15) main() {  
(16)     char x;  
(17)     ...  
(18) }
```

The Symbol Table

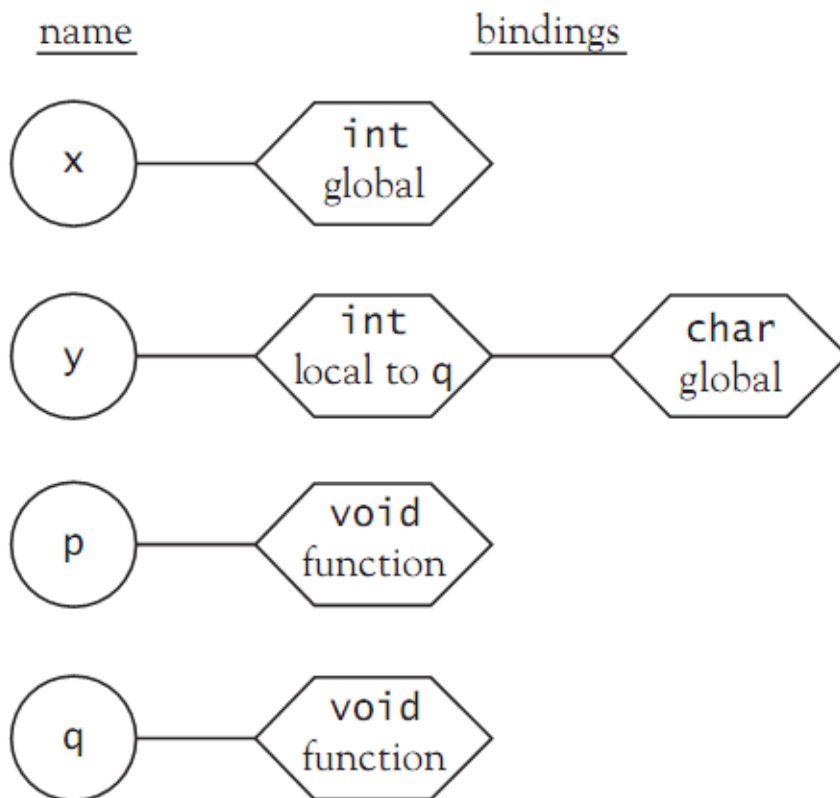


Figure 7.10 Symbol table structure at line 13 of Figure 7.6

```
(1) int x;  
(2) char y;  
  
(3) void p() {  
(4)     double x;  
(5)     ...  
(6)     { int y[10];  
(7)         ...  
(8)     }  
(9)     ...  
(10) }  
  
(11) void q() {  
(12)     int y;  
  
(13)     ...  
(14) }  
  
(15) main() {  
(16)     char x;  
(17)     ...  
(18) }
```

The Symbol Table

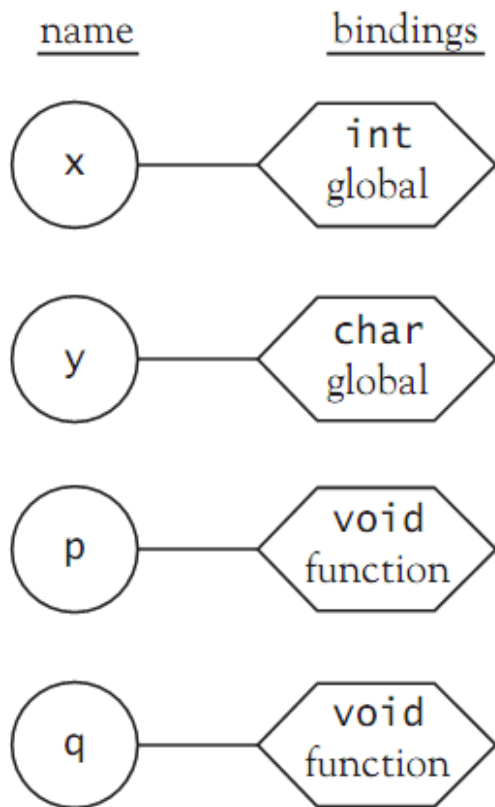
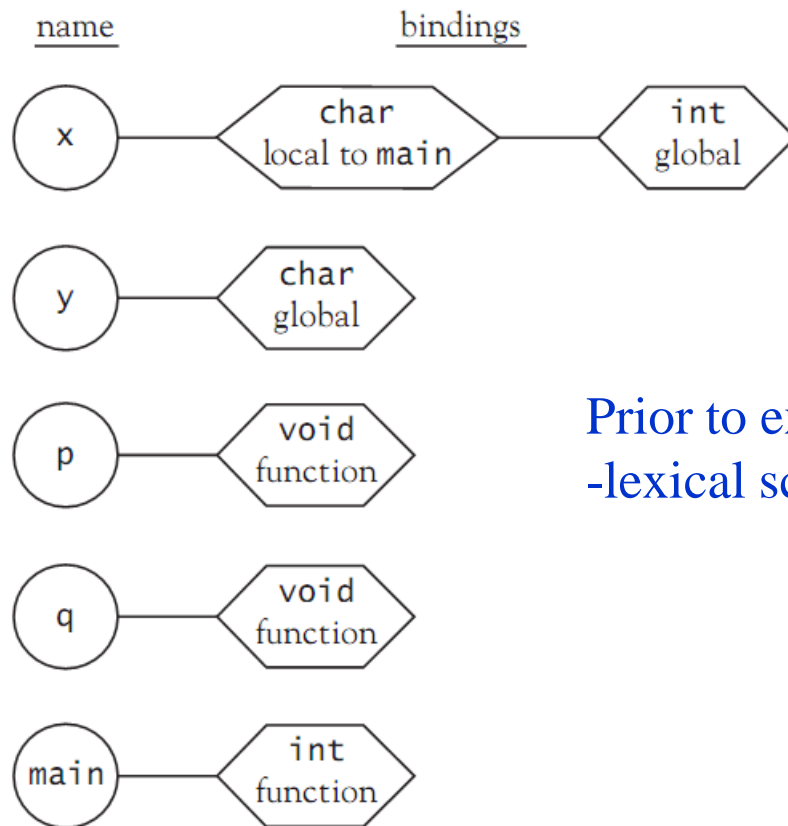


Figure 7.11 Symbol table structure at line 14 of Figure 7.6

```
(1) int x;  
(2) char y;  
  
(3) void p() {  
(4)     double x;  
(5)     ...  
(6)     { int y[10];  
(7)         ...  
(8)     }  
(9)     ...  
(10) }  
  
(11) void q() {  
(12)     int y;  
  
(13)     ...  
(14) }  
  
(15) main() {  
(16)     char x;  
(17)     ...  
(18) }
```

The Symbol Table



Prior to execution
-lexical scope

```
(1) int x;  
(2) char y;  
  
(3) void p(){  
(4)     double x;  
(5)     ...  
(6)     { int y[10];  
(7)         ...  
(8)     }  
(9)     ...  
(10) }
```

```
(11) void q(){  
(12)     int y;
```

```
(13)     ...  
(14) }
```

```
(15) main(){  
(16)     char x;  
(17)     ...  
(18) }
```

Figure 7.12 Symbol table structure at line 17 of Figure 7.6

The Symbol Table

The previous example assumes that declarations are processed statically (prior to execution)

- This is called **static scoping** or **lexical scoping**
- Symbol table is managed by a compiler
- Bindings of declarations are all static

If symbol table is managed dynamically (during execution), declarations will be processed as they are encountered along an execution path

- This is called **dynamic scoping**

```
(1) #include <stdio.h>

(2) int x = 1;
(3) char y = 'a';
(4) void p(){
(5)     double x = 2.5;
(6)     printf("%c\n",y);
(7)     { int y[10];
(8)     }
(9) }

(10) void q(){
(11)     int y = 42;
(12)     printf("%d\n",x);
(13)     p();
(14) }

(15) main(){
(16)     char x = 'b';
(17)     q();
(18)     return 0;
(19) }
```

Figure 7.13 C program of Figure 7.6 with added code

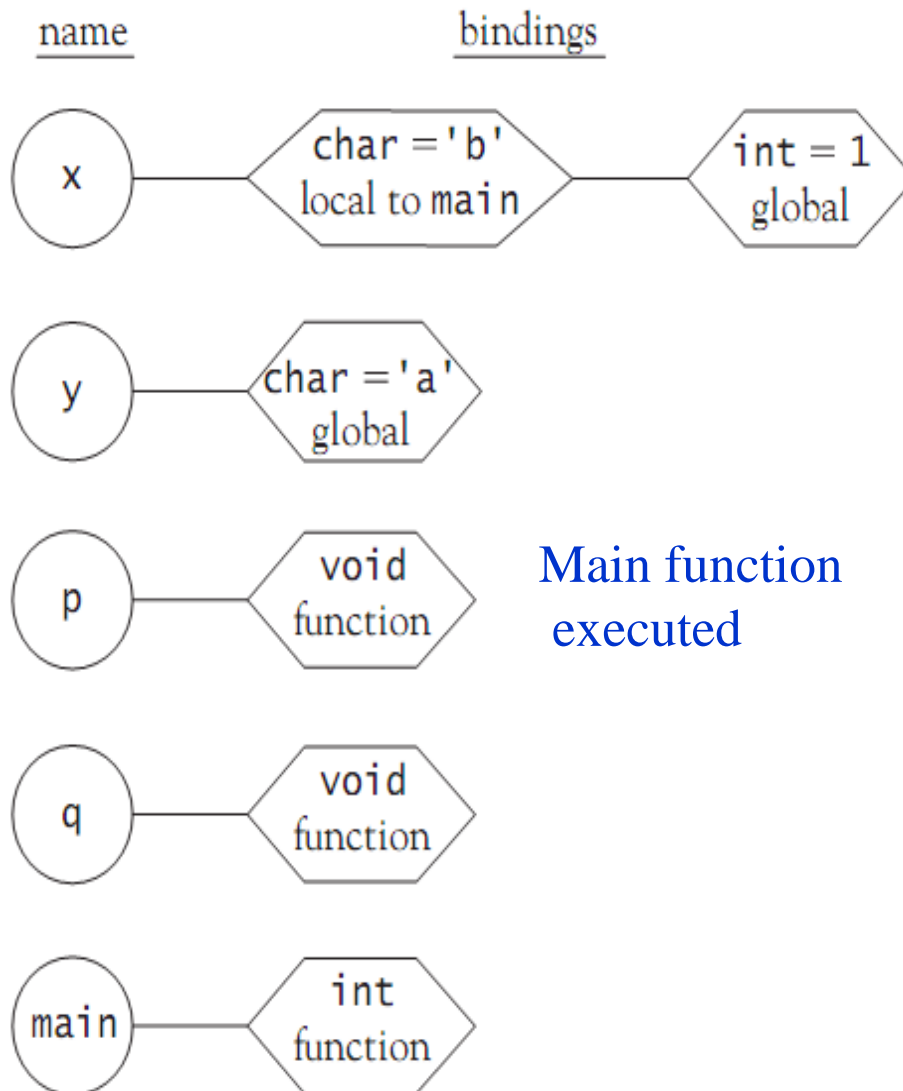


Figure 7.14 Symbol table structure at line 17 of Figure 7.13 using dynamic scope

```

(1) #include <stdio.h>

(2) int x = 1;
(3) char y = 'a';
(4) void p(){
(5)     double x = 2.5;
(6)     printf("%c\n",y);
(7)     { int y[10];
(8)     }
(9) }

(10) void q(){
(11)     int y = 42;
(12)     printf("%d\n",x);
(13)     p();
(14) }

(15) main(){
(16)     char x = 'b';
(17)     q();
(18)     return 0;
(19) }

```

Figure 7.13 C program of Figure 7.6 with added code

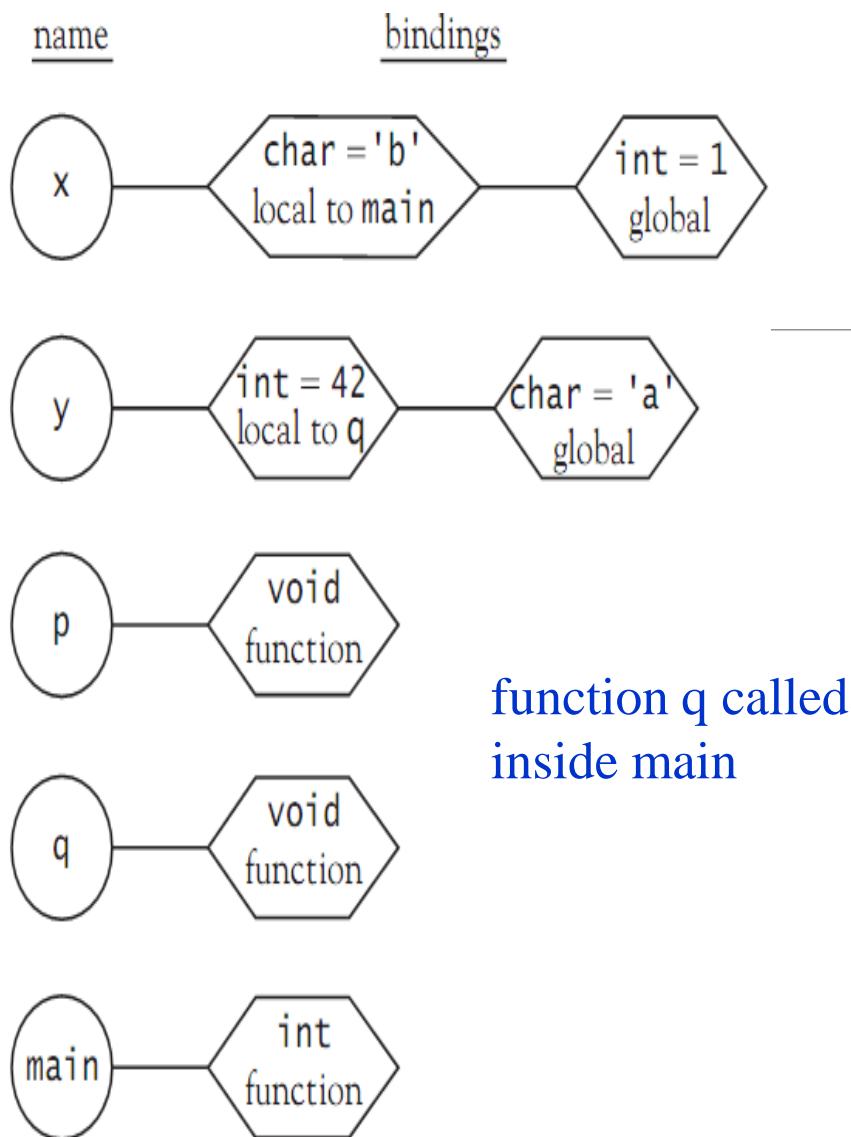


Figure 7.15 Symbol table structure at line 12 of Figure 7.13 using dynamic scope

```

(1) #include <stdio.h>

(2) int x = 1;
(3) char y = 'a';
(4) void p(){
(5)     double x = 2.5;
(6)     printf("%c\n",y);
(7)     { int y[10];
(8)     }
(9) }

(10) void q(){
(11)     int y = 42;
(12)     printf("%d\n",x);
(13)     p();
(14) }

(15) main(){
(16)     char x = 'b';
(17)     q();
(18)     return 0;
(19) }

```

Figure 7.13 C program of Figure 7.6 with added code

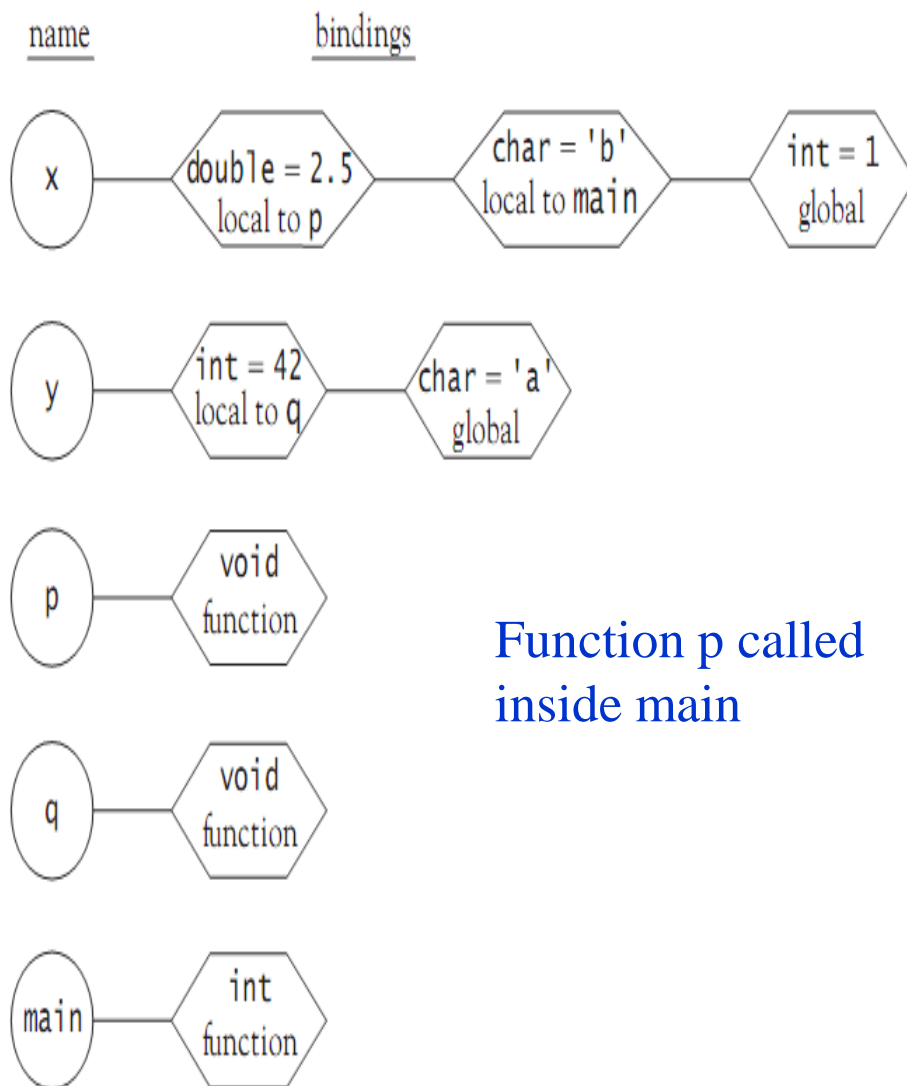


Figure 7.16 Symbol table structure at line 6 of Figure 7.13 using dynamic scope

```

(1) #include <stdio.h>

(2) int x = 1;
(3) char y = 'a';
(4) void p() {
(5)     double x = 2.5;
(6)     printf("%c\n", y);
(7)     { int y[10];
(8)     }
(9) }

(10) void q() {
(11)     int y = 42;
(12)     printf("%d\n", x);
(13)     p();
(14) }

(15) main() {
(16)     char x = 'b';
(17)     q();
(18)     return 0;
(19) }

```

Figure 7.13 C program of Figure 7.6 with added code

The Symbol Table

Dynamic scoping will affect the semantics of the program and produce different output

Output using lexical scoping:

1

a

Output using dynamic scoping:

98

*

Dynamic scope can be problematic, which is why few languages use it

The Symbol Table

Problems with dynamic scoping:

- The declaration of a nonlocal name cannot be determined by simply reading the program: the program must be executed to know the execution path
- Since nonlocal variable references cannot be predicted prior to execution, neither can their data types

Dynamic scoping is a possible option for highly dynamic, interpreted languages when programs are not expected to be extremely large

The Symbol Table

Runtime environment is simpler with dynamic scoping in an interpreter

- APL, Snobol, Perl, and early dialects of Lisp were dynamically scoped
- Scheme and Common Lisp use static scoping

There is additional complexity for symbol tables

`struct` declaration must contain further declarations of the data fields within it

- Those fields must be accessible using dot member notation whenever the `struct` variable is in scope

The Symbol Table

Two implications for struct variables:

- A struct declaration actually contains a local symbol table itself as an attribute
- This local symbol table cannot be deleted until the struct variable itself is deleted from the global symbol table of the program

```

(1) struct{
(2)     int a;
(3)     char b;
(4)     double c;
(5) } x = {1, 'a', 2.5};

(6) void p(){
(7)     struct{
(8)         double a;
(9)         int b;
(10)        char c;
(11)    } y = {1.2, 2, 'b'};
(12)    printf("%d, %c, %g\n", x.a, x.b, x.c);
(13)    printf("%f, %d, %c\n", y.a, y.b, y.c);
(14) }

(15) main(){
(16)     p();
(17)     return 0;
(18) }

```

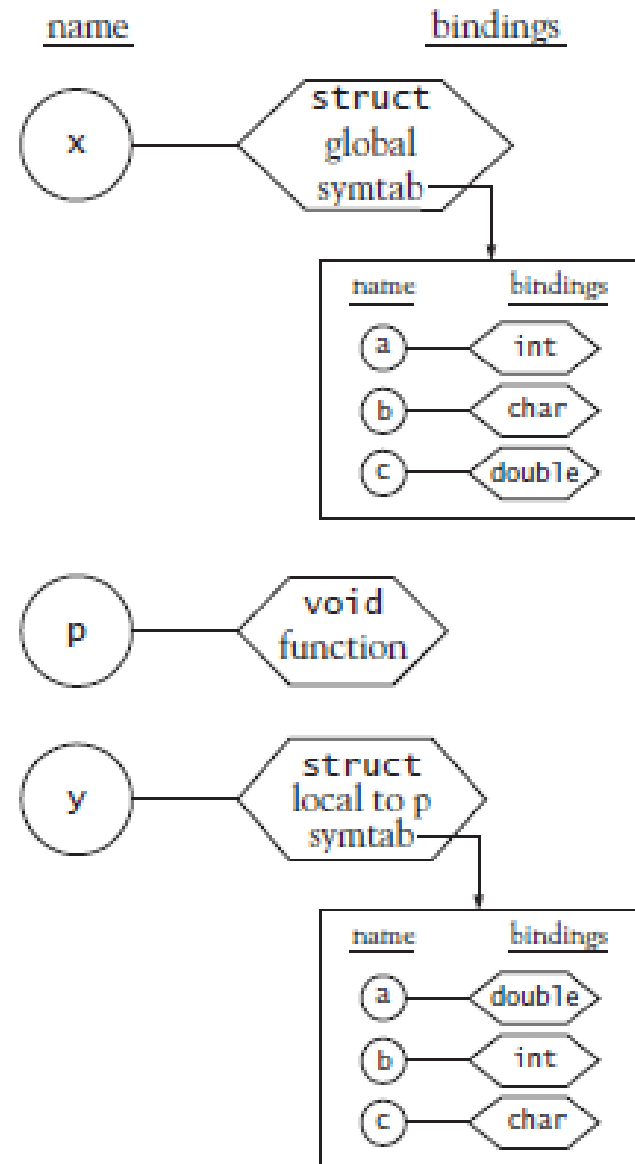


Figure 7.17 Code example illustrating scope of local declarations in a C struct

The Symbol Table

Any scoping structure that can be referenced directly must also have its own symbol table

Examples:

- Named scopes in Ada
- Classes, structs, and namespaces in C++
- Classes and packages in Java

Typically, there will be a table for each scope in a stack of symbol tables

- When a reference to a name occurs, a search begins in the current table and continues to the next table if not found, and so on

```

(1) with Text_IO; use Text_IO;
(2) with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

(3) procedure ex is
(4)   x: integer := 1;
(5)   y: character := 'a';

(6)   procedure p is
(7)     x: float := 2.5;
(8)   begin
(9)     put(y); new_line;
(10)  A: declare
(11)    y: array (1..10) of integer;
(12)  begin
(13)    y(1) := 2;
(14)    put(y(1)); new_line;
(15)    put(ex.y); new_line;
(16)  end A;
(17) end p;

(18) procedure q is
(19)   y: integer := 42;
(20) begin
(21)   put(x); new_line;
(22)   p;
(23) end q;

(24) begin
(25) declare
(26)   x: character := 'b';
(27) begin
(28)   q;
(29)   put(ex.x); new_line;
(30) end;
(31) end ex;

```

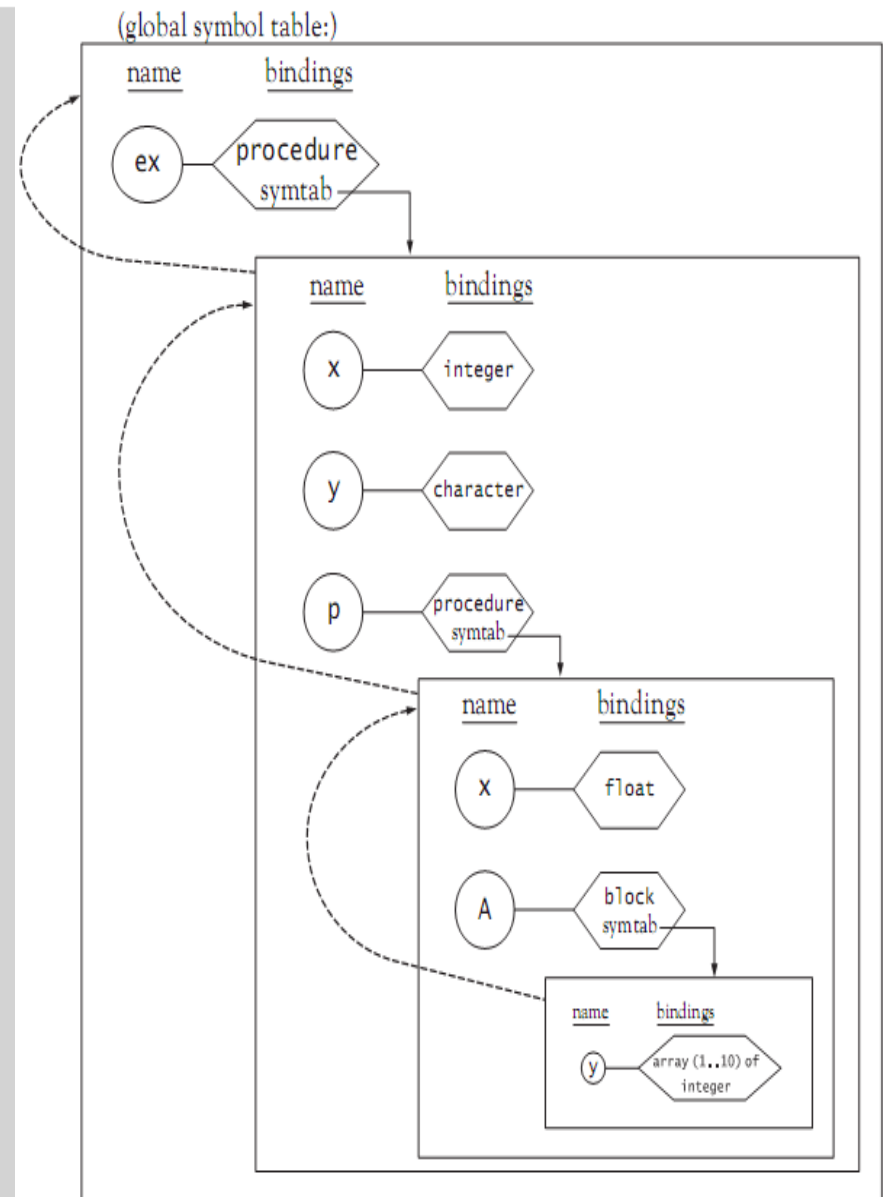


Figure 7.20 Symbol table structure at line 12 of Figure 7.19

Symbol Table Activity

Name Resolution and Overloading

Addition operator + actually indicates at least two different operations: integer addition and floating-point addition

- + operator is said to be **overloaded**

Translator must look at the data type of each operand to determine which operation is indicated

Overload resolution: process of choosing a unique function among many with the same name

- Lookup operation of a symbol table must search on name plus number and data type of parameters

Name Resolution and Overloading

```
int max(int x, int y){ // max #1
    return x > y ? x : y;
}

double max(double x, double y){ // max #2
    return x > y ? x : y;
}

int max(int x, int y, int z){ // max #3
    return x > y ? (x > z ? x : z) : (y > z ? y : z);
}
```

Figure 7.21 Three overloaded `max` functions in C++

Name Resolution and Overloading

Consider these function calls:

```
max(2,3); // calls max #1
```

```
max(2.1,3.2); // calls max #2
```

```
max(1,3,2); // calls max #3
```

Symbol table can determine the appropriate function based on number and type of parameters

Calling context: the information contained in each call

But this **ambiguous** call depends on the language rules (if any) for converting between data types:

```
max(2.1,3); // which max?
```


Name Resolution and Overloading

Adding these definitions makes the function calls legal in C++ and Ada but is unnecessary in Java

```
double max(int x, double y) { // max #4
    return x > y ? (double) x : y;
}

double max(double x, int y) { // max #5
    return x > y ? x : (double) y;
}
```

Autc **Figure 7.22** Two more overloaded max functions in C++ (see Figure 7.21) complicate overload resolution

Name Resolution and Overloading

Additional information in a calling context may be used for overload resolution:

- Ada allows the return type and names of parameters to be used for overload resolution
- C++ and Java ignore the return type

Both Ada and C++ (but not Java) allow built-in operators to be overloaded

When overloading a built-in operator, we must accept its syntactic properties

- Example: cannot change the associativity or precedence of the + operator

Name Resolution and Overloading

Note that there is no semantic difference between operators and functions, only syntactic difference

- Operators are written in infix form
- Function calls are always written in prefix form

Names can also be overloaded

Some languages use different symbol tables for each of the major kinds of definitions to allow the same name for a type, a function, and a variable

- Example: Java

Allocation, Lifetimes, and the Environment

Environment: maintains the bindings of names to locations

- May be constructed statically (at load time), dynamically (at execution time), or with a mixture of both

Not all names in a program are bound to locations

- Examples: names of constants and data types may represent purely compile-time quantities

Declarations are also used in environment construction

- Indicate what allocation code must be generated

Allocation, Lifetimes, and the Environment

Typically, in a block-structured language:

- Global variables are allocated statically
- Local variables are allocated dynamically when the block is entered

When a block is entered, memory for variables declared in that block is allocated

When a block is exited, this memory is deallocated

```

(1) A: { int x;
(2)      char y;
(3)      ...
(4)    B: { double x;
(5)          int a;
(6)          ...
(7)        } /* end B */
(8)    C: { char y;
(9)          int b;
(10)         ...
(11)       D: { int x;
(12)             double y;
(13)             ...
(14)           } /* end D */
(15)         ...
(16)       } /* end C */
(17)     ...
(18)   } /* end A */

```

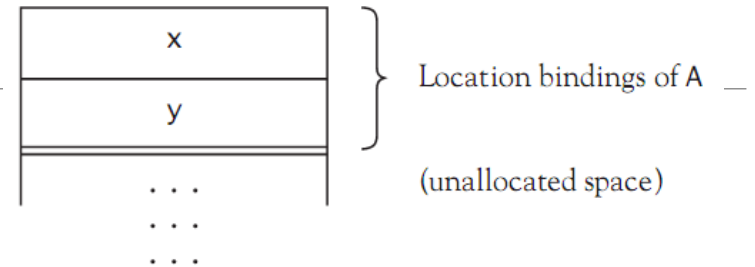


Figure 7.28 The environment at line 3 of Figure 7.27 after the entry into A

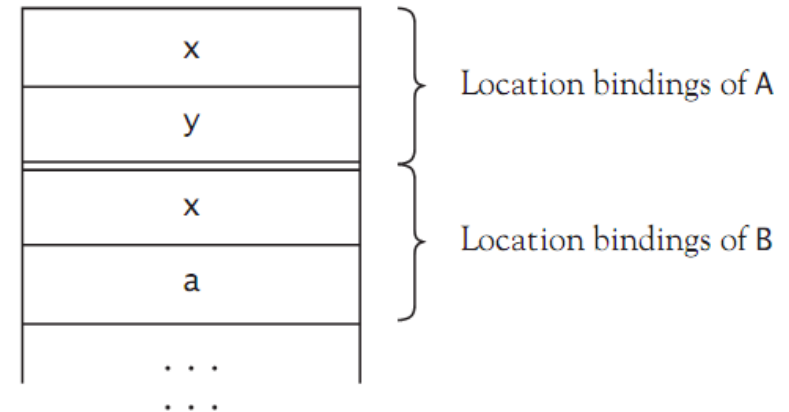


Figure 7.29 The environment at line 6 of Figure 7.27 after the entry into B

```

(1) A: { int x;
(2)      char y;
(3)      ...
(4)    B: { double x;
(5)          int a;
(6)          ...
(7)      } /* end B */
(8)    C: { char y;
(9)          int b;
(10)         ...
(11)      D: { int x;
(12)              double y;
(13)              ...
(14)          } /* end D */
(15)         ...
(16)      } /* end C */
(17)      ...
(18)  } /* end A */

```

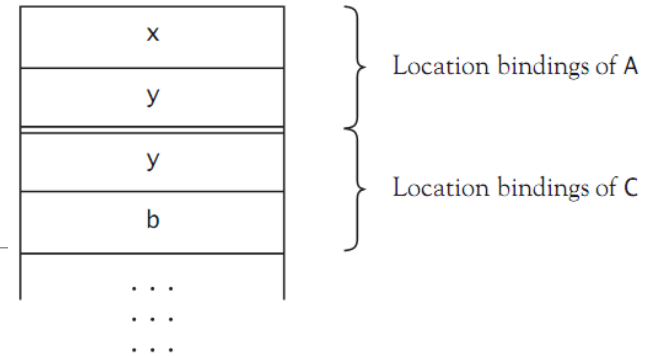


Figure 7.30 The environment at line 10 of Figure 7.27 after the entry into C

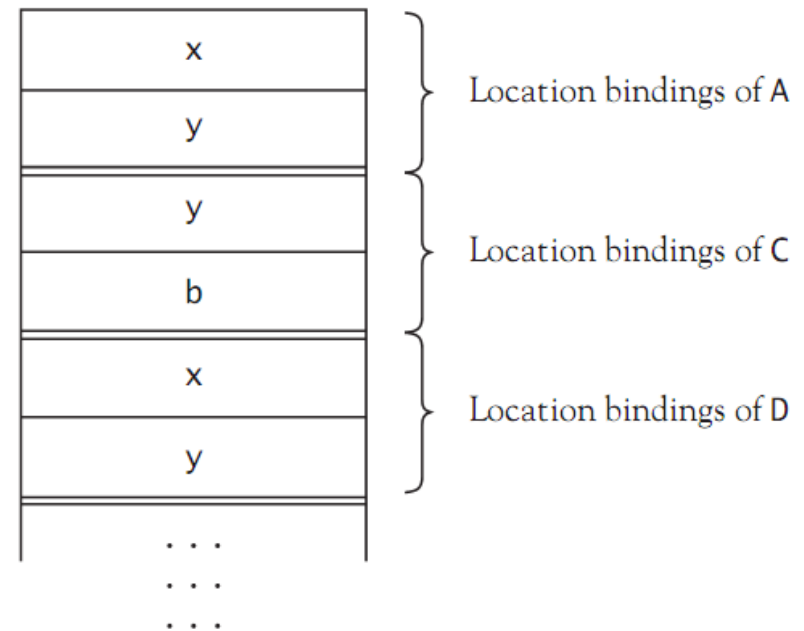


Figure 7.31 The environment at line 1 of Figure 7.27 after the entry into D

Allocation, Lifetimes, and the Environment

Memory for local variables within a function will not be allocated until the function is called

Activation: a call to a function

Activation record: the corresponding region of allocated memory

In a block-structured language with lexical scope, the same name can be associated with different locations, but only one of these can be accessed at any one time

Lifetime (or **extent**) of an object is the duration of its allocation in the environment

Allocation, Lifetimes, and the Environment

Lifetime of an object can extend beyond the region of a program in which it can be accessed

- Lifetime extends through a scope hole

Pointer: an object whose stored value is a reference to another object

C allows the initialization of pointers that do not point to an allocated object:

- Objects must be manually allocated by use of an allocation routine
- Variable can be dereferenced using the unary * operator `int* x = NULL;`

Allocation, Lifetimes, and the Environment

C++ simplifies dynamic allocation with operators `new` and `delete`:

```
int* x = new int; // C++
*x = 2;
cout << *x << endl; // output in C++
delete x;
```

- These are used as unary operators, not functions

Heap: area in memory from which locations can be allocated in response to calls to `new`

Dynamic allocation: allocation on the heap

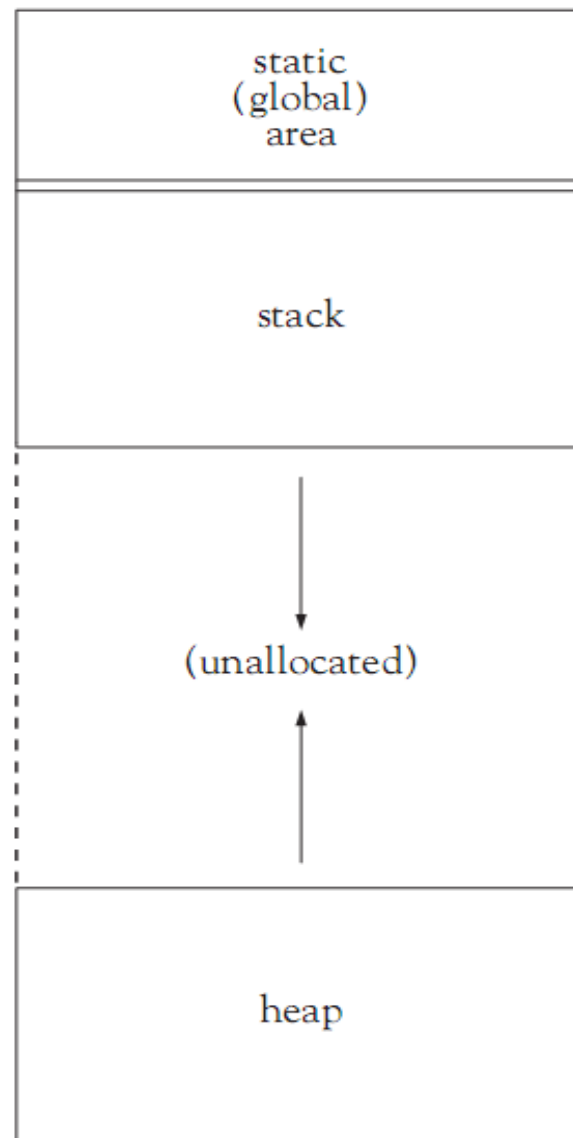


Figure 7.32 Structure of a typical environment with a stack and a heap

Allocation, Lifetimes, and the Environment

Many languages require that heap deallocation be managed automatically

Heap allocation/deallocation and explicit pointer manipulation are inherently unsafe operations

- Can introduce seriously faulty runtime behavior that may even compromise the operating system

Storage class: the type of allocation

- Static (for global variables)
- Automatic (for local variables)
- Dynamic (for heap allocation)

Variables and Constants

Although references to variables and constants look the same in many languages, their roles and semantics are very different

We will look at the basic semantics of both

Variables

Variable: an object whose stored value can change during execution

- Is completely specified by its attributes (name, location, value, data type, size of memory storage)

Box and circle diagram: focuses on name and location

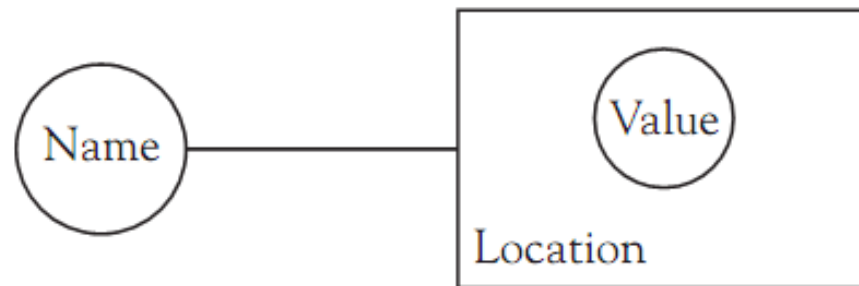


Figure 7.36 Schematic representation of a variable, its value, and its location

Variables

Assignment statement: principle way in which a variable changes its value

Example: $x = e;$

- Semantics: expression e is evaluated to a value, then copied into the location of x

If e is a variable named y :

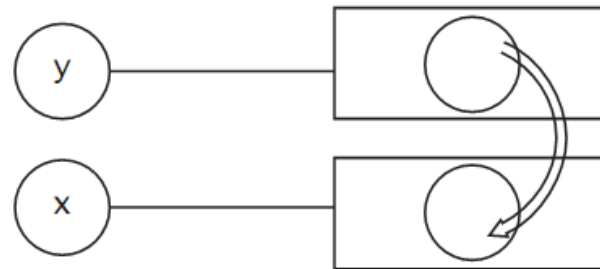


Figure 7.37 Assignment with copying of values

Variables

Variable on right side of assignment statement stands for a value (**r-value**); variable on left side stands for a location (**l-value**)

Address of operator (&) in C: turns a reference into a pointer to fetch the address of a variable

Assignment by sharing: the location is copied instead of the value

Assignment by cloning: allocates new location, copies value, and binds to the new location

Both are sometimes called **pointer semantics** or **reference semantics**

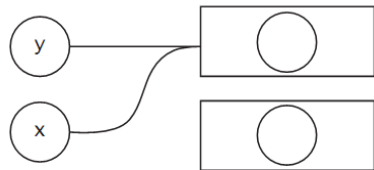


Figure 7.38 The result of assignment by sharing

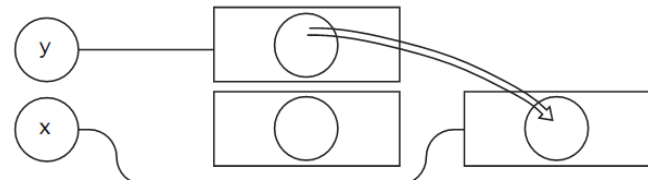


Figure 7.39 The result of assignment by cloning

Variables

Storage semantics or **value semantics** refer to standard assignment

Standard implementation of assignment by sharing uses pointers and implicit dereferencing

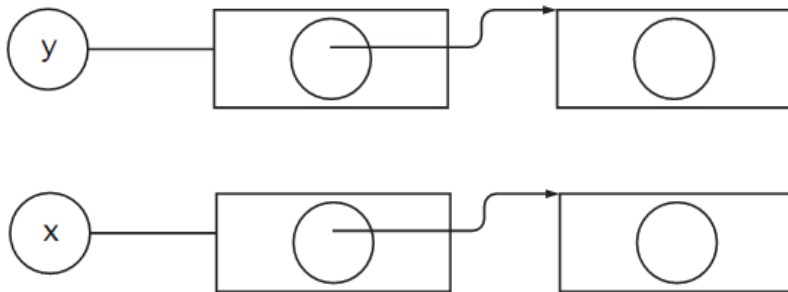


Figure 7.40 Variables as implicit references to objects

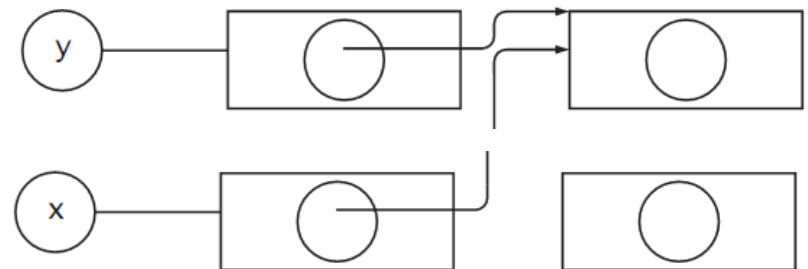


Figure 7.41 Assignment by sharing of references

Constants

Constant: an entity with a fixed value for the duration of its existence in a program

- Like a variable, but has no location attribute
- Sometimes say that a constant has **value semantics**

Literal: a representation of characters or digits

Compile-time constant: its value can be computed during compilation

Static constant: its value can be computed at load time

Constants

Manifest constant: a name for a literal

Dynamic constant: its value must be computed during execution

Function definitions in virtually all languages are definitions of constants whose values are functions

- This differs from a function variable in C, which must be defined as a pointer

Constants

a and b are compile-time constants

- a is a manifest constant

c is a static (load-time constant)

d is a dynamic constant

```
#include <stdio.h>
#include <time.h>

const int a = 2;
const int b = 27+2*2;
/* warning - illegal C code! */
const int c = (int) time(0);

int f( int x){
    const int d = x+1;
    return b+c;
}

...
```

Aliases, Dangling References, and Garbage

There are several problems with naming and dynamic allocation conventions of programming languages, especially C, C++, and Ada

As a programmer, you can learn to avoid those problematic situations

As a language designer, you can build solutions into your language

Aliases

Alias: occurs when the same object is bound to two different names at the same time

Can occur during procedure call, through the use of pointer variables, or through assignment by sharing

```
(1) int *x, *y;  
(2) x = (int *) malloc(sizeof(int));  
(3) *x = 1;  
(4) y = x;    /* *x and *y now aliases */  
(5) *y = 2;  
(6) printf("%d\n", *x);
```

Aliases

```
(1) int *x, *y;  
(2) x = (int *) malloc(sizeof(int));  
(3) *x = 1;  
(4) y = x;    /* *x and *y now aliases */  
(5) *y = 2;  
(6) printf("%d\n", *x);
```

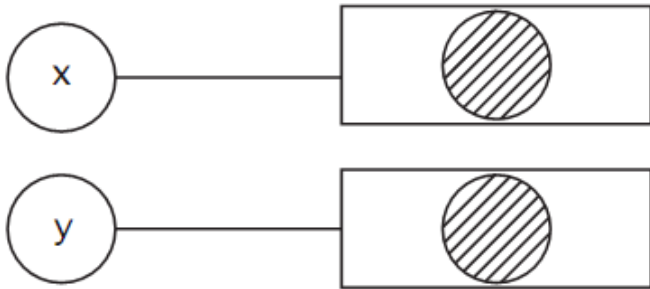


Figure 7.45 Allocation of storage for pointers x and y

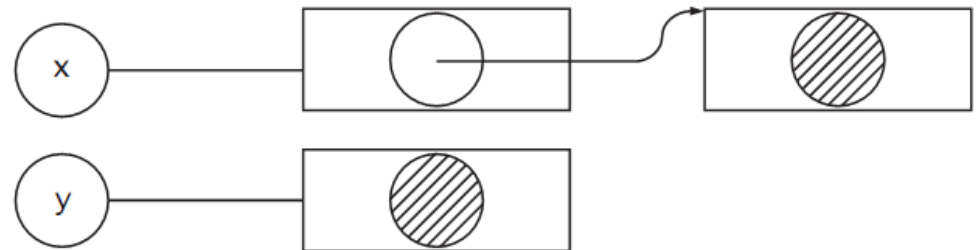


Figure 7.46 Allocation of storage for *x

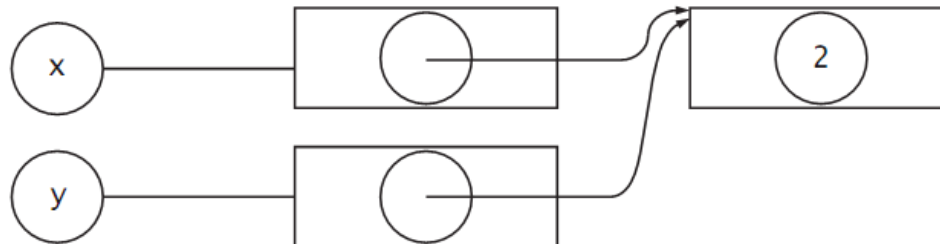


Figure 7.49 Result of *y = 2

Aliases

Aliases can potentially cause harmful side effects

Side effect: any change in a variable's value that persists beyond the execution of the statement

Not all side effects are harmful; an assignment statement is intended to cause one

Side effects that change variables whose names do not directly appear in the statement are potentially harmful

- Cannot be determined from the written code

Aliasing due to pointer assignment is difficult to control

Aliases

Assignment by sharing implicitly uses pointers

Java has a mechanism for explicitly **cloning** an object so that aliases are not created by assignment

```
(1) class ArrTest{  
(2)     public static void main(String[] args){  
(3)         int[] x = {1,2,3};  
(4)         int[] y = x;  
(5)         x[0] = 42;  
(6)         System.out.println(y[0]);  
(7)     }  
(8) }
```

Dangling References

Dangling reference: a location that has been deallocated from the environment but can still be accessed by a program

- Occurs when a pointer points to a deallocated object

```
int *x , *y;
...
x = (int *) malloc(sizeof(int));
...
*x = 2;
...
y = x; /* *y and *x now aliases */
free(x); /* *y now a dangling reference */
...
printf("%d\n",*y); /* illegal reference */
```

Dangling References

Can also result from automatic deallocation of local variables on exit from a block, with the C address of operator

```
(1) { int * x;  
(2)   { int y;  
(3)     y = 2;  
(4)     x = &y;  
(5)   }  
(6)   /* *x is now a dangling reference */  
(7) }
```

Garbage

Garbage: memory that has been allocated in the environment but is now inaccessible to the program

Can occur in C by failing to call free before reassigning a pointer variable

```
int *x;  
...  
x = (int *) malloc(sizeof(int));  
x = 0;
```

A program that is internally correct but produces garbage may run out of memory

Garbage

A program with dangling references may:

- Produce incorrect results
- Corrupt other programs in memory
- Cause runtime errors that are hard to locate

For this reason, it is useful to remove the need to deallocate memory explicitly from the programmer

Garbage collection: process of automatically reclaiming garbage

Language design is a key factor in the kind of runtime environment necessary for correct execution of programs