Semantics

Basic Semantics

- Syntax tells what the language constructs look like.
- Semantics tells what the language constructs actually do.

Different ways to specify semantics

- By a language reference manual.
- By a defining translator.
- By a formal definition

By a language reference manual.

- Most common way to specify semantics.
- The expanding use of English descriptions has resulted in clearer and more precise reference manuals

Drawbacks:

- Lack of precision inherent in natural language descriptions.
- May have omissions and ambiguities

By a defining translator

Advantage

- Question about the language can be answered by experiment. (Like in physics and chemistry)

By a defining translator

Drawbacks

- Question about program behavior cannot be answered in advance.(Execution is necessary)
- Bugs and machine dependencies in the translator become parts of the language semantic.
- Translators may not be portable to all machines.

By a formal definition

- Formal, mathematical methods are precise, but they are also complex and abstract, and require study to understand.
- Best formal method to use for the description of the translation and execution of programs is denotational semantics, which describes semantics using a series of functions.
- Advantage :- It is precise.

Attributes, Binding and semantic functions

- A fundamental abstraction mechanism in a programming language is the use of **names, or identifiers,** to denote language entities or constructs.

Attributes, Binding and semantic functions

- In most languages variables, procedures and constants can have the names assigned by the programmer.
- Semantics describes meaning of each name used in program.

Attributes, Binding and semantic functions

- Semantics not only describes names but also location and value
- Values are any storable quantities, such as the integers, the reals, or even array values consisting of a sequence of the values stored at each index of the array.

Attributes, Binding and semantic functions

- Locations are places where values can be stored.
- Locations are like addresses in the memory of a specific computer.

Attributes, Binding and semantic functions

- The meaning of a name is determined by the properties, or **attributes,** associated with the name.
- For example, the C declaration

  const int n = 5;

  makes n into an integer constant with value 5.
- The C declaration

  int x; associates the attribute “variable” and the data type “integer” to the name x.

Attributes, Binding and semantic functions

The C declaration

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

associates the attribute “function” and the following additional attributes to the name f:

- 1. The number, names, and data types of its parameters (in this case, one parameter with name n and data type “integer”)
- 2. The data type of its returned value (in this case, “double”)
- 3. The body of code to be executed when f is called (in this case, we have not written this code but just indicated it with three dots)

Attributes, Binding and semantic functions

- Declarations are not the only language constructs that can associate attributes to names.

  For example, the assignment:

  ```
  x = 2;
  ```

  associates the new attribute “value 2” to the variable x. And, if y is a pointer variable declared as:

  ```
  int* y;
  ```

  allocates memory for an integer variable (that is, associates a location attribute to it) and assigns this location to *y, that is, associates a new value attribute to y.

Attributes

- Properties of language entities, especially identifiers used in a program.
- Important examples:
  - Value of an expression
  - Data type of an identifier
  - Maximum number of digits in an integer
  - Location of a variable
  - Code body of a function or method
- Declarations ("definitions") bind attributes to identifiers.

Binding

- The process of associating an attribute to a name is called binding.

Binding Time

- Binding time will tell when the attribute is computed and bound to a name.

Binding Time

- Binding time can be classified into two general categories.
- Static Binding(Occurs prior to execution)
- Dynamic Binding (Occurs during execution)

Binding Time

- Functional languages have more dynamic binding than imperative languages
- Interpreters, for example, can translate and execute code simultaneously and so can establish most bindings dynamically, while compilers can perform many bindings statically.

Binding Time

- As examples of binding times, consider the examples of attributes. In the declaration

  const int n = 2;

  the value 2 is bound statically to the name n, and in the declaration

  int x;

  the data type “integer” is bound statically to the name x.
- On the other hand, the assignment x = 2 binds the value 2 dynamically to x when the assignment statement is executed.
- And the C++ statement y = new int;

  dynamically binds a storage location to *y and assigns that location as the value of y.

Possible binding times

- Language definition time.(Pascal boolean)
- Language implementation time.(maxint in C).
- Translation time(data type bound to name)
- Link time(body of an externally defined function)
- Load time(location of a global variable)
- Execution time(allocation of memory to identifier)

Symbol Table

- Symbol Table is a function that expresses binding of attributes to names.
- Symbol Table: Names → attributes

  or more graphically as shown in the figure

Figure 7.1 Mapping names to attributes in a symbol table

Symbol Table

- During the execution of a compiled program, attributes such as locations and values must also be maintained. A compiler generates code that maintains these attributes in data structures during execution.
- The memory allocation part of this process—that is, the binding of names to storage locations is usually considered separately and is called the **environment,** as shown in Figure.

Figure 7.2 Mapping names to locations in an environment

Symbol Table

- Finally, the bindings of storage locations to values is called the **memory,** since it abstracts the memory of an actual computer (sometimes it is also called the **store** or the **state**) as shown in the figure.

Figure 7.3 Mapping locations to values in a memory

Declarations

- Declaration is primary method for establishing bindings.
- Bindings can be determined by a declaration either implicitly or explicitly.

Declarations

- int x;
- Establishes the data type of x explicitly using the keyword int.
- Exact location of x during execution is only bound implicitly and it can be either static or dynamic.
- Similarly value of x is either implicitly zero or undefined depending on the location of the declaration.

Declarations

- Entire declaration can be implicit.
- Example : all variables in Fortran that are not explicitly declared are assumed to be integer if their name begin with “I”, “J”, “K”, “L”, “M”, or “N” and real otherwise.
- In Basic variables ending in “%” are integers, Variables ending in “$” are strings and all other are real

Declarations

- Declarations that bind all potential attributes are called definitions while declarations that only partially specify attributes are called declarations.
- For example, the function declaration (or **prototype)**

  double f(int x);

  specifies only the data type of the function f (i.e., the types of its parameters and return value) but does not specify the code to implement f.

Declarations

Similarly,

```
struct x;
```

specifies an **incomplete type** in C or C++, so this declaration is also not a definition.

Block

- A block consists of a sequence of declarations followed by sequence of statements and surrounded by syntactic markers such as braces or begin end pairs.

Block

- In C, blocks are called **compound statements and** appear as the body of functions in function definitions, and also anywhere an ordinary program statement could appear.

```
void p (){
double r, z; /* the block of p */
...
{ int x, y; /* another, nested, block */
x = 2;
y = 0;
x += 1;
}
...
}
```

Block

- Declarations that are associated with a specific block are called **local** while declaration in surrounding blocks are called **non local** declarations.

- For example, in the previous definition of the procedure p, variables r and z are local to p but are nonlocal from within the second block nested inside p.

Scope of Binding

- The scope of binding is the region of the program over which the binding is maintained.
- In a block structured language like C where block can be nested the scope of the binding is limited to the block in which its associated declaration appears.
- Such a scope rule is called lexical scope.

Scope of Binding

```
int x;
void p(){
        char y;
        …
 } /* p */
void q(){
        double z;
        …
} /* q */
main(){
        int w[10];
        ...
}
```

Scope of Binding

- The declarations of variable x (line 1) and procedures p (lines 2–5), q (lines 6–9), and main (lines 10–13) are global.

- The declarations of y (line 3), z (line 7), and w (line 11), on the other hand, are associated with the blocks of procedures p, q, and main, respectively. They are local to these functions, and their declarations are valid only for those functions.

Scope of Binding

- Declaration in nested blocks takes precedence over global declarations.

```
int x;
void p(){
    char x;
    x = 'a'; /* assigns to char x */
    ...
}
main(){
    x = 2; /* assigns to global x */
    ...
}
```

Scope of Binding

- The declaration of x in p (line-3) takes precedence over the global declaration of x (line-1) in the body of p.

- Thus, the global integer x cannot be accessed from within p.

- The global declaration of x is said to have a **scope hole** inside p, and the local declaration of x is said to **shadow** its global declaration.

Visibility of declarations

- For this reason, a distinction is sometimes made between the scope and the visibility of a declaration.
- Visibility includes only those regions of a program where the bindings of the declaration apply while scope includes scope holes.

Visibility of declarations

- In C++, the **scope resolution operator ::** (double colon) can be used to access such hidden declarations (as long as they are global).

  For example:

```
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
        ...
}
```

Lexical Vs. Dynamic Scope

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

Java scope example

```
public class Scope
  { public static int x = 2;
    public static void f()
    { System.out.println(x); }
    public static void main(String[] args)
    { int x = 3;
      f();
    }
  }
```

- Of course, this prints 2, but under dynamic scope it would print 3 (the most recent declaration of `x` in the execution path is found).

Dynamic scope evaluated

- Almost all languages use lexical scope: with dynamic scope the meaning of a variable cannot be known until execution time, thus there cannot be any static checking.
- In particular, no static type checking.
- Originally used in Lisp. Scheme could still use it, but doesn't. Some languages still use it: Javascript (supported but not the default), Perl (older versions, newer versions have both).
- Lisp inventor (McCarthy) now calls it a bug.
- Still useful as a pedagogical tool to understand the workings of scope. In some ways a lot like dynamic binding of methods.

Scope holes

- Under either lexical or dynamic scope, a nested or more recent declaration can mask a prior declaration. Indeed, in slide 41, the local declaration of `x` in `main` masks the static declaration of `x` in the `Scope` class.

- How would you access the static `x` inside `main` in Java?
- Use `Scope.x` in place of `x`:

```
public static void main(String[] args)
{ int x = 3; Scope.x = 4; ...}
```

Symbol Table

- A symbol table is like a dictionary. It must support insertion, lookup and deletion of names with associated attributes, representing the bindings in declarations.
- Symbol table can be maintained by number of data structures such as hash tables, trees, lists.
- However, the maintenance of scope information in a lexically scoped language with block structure requires that declarations be processed in a stacklike fashion.
- That is, on entry into a block, all declarations of that block are processed and the corresponding bindings added to the symbol table.
- Then, on exit from the block, the bindings provided by the declarations are removed, restoring any previous bindings that may have existed.
- This process is called **scope analysis.**

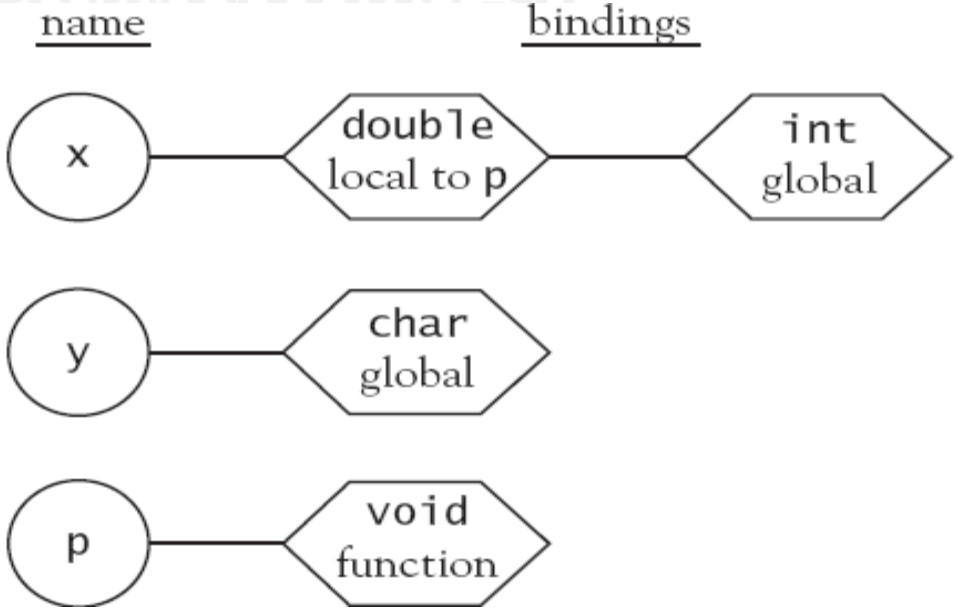
Symbol Table

- We can view symbol table as collection of names each of which has a stack of declarations associated with it such that declaration on top of the stack is the one whose scope is currently active.

Symbol Table Structure

```
int x;
char y;
void p(){
    double x;
    ...
    { int y[10];
    ...
    }
    ...
}
void q(){
    int y;
    ...
}
main(){
    char x;
    ...
}
```

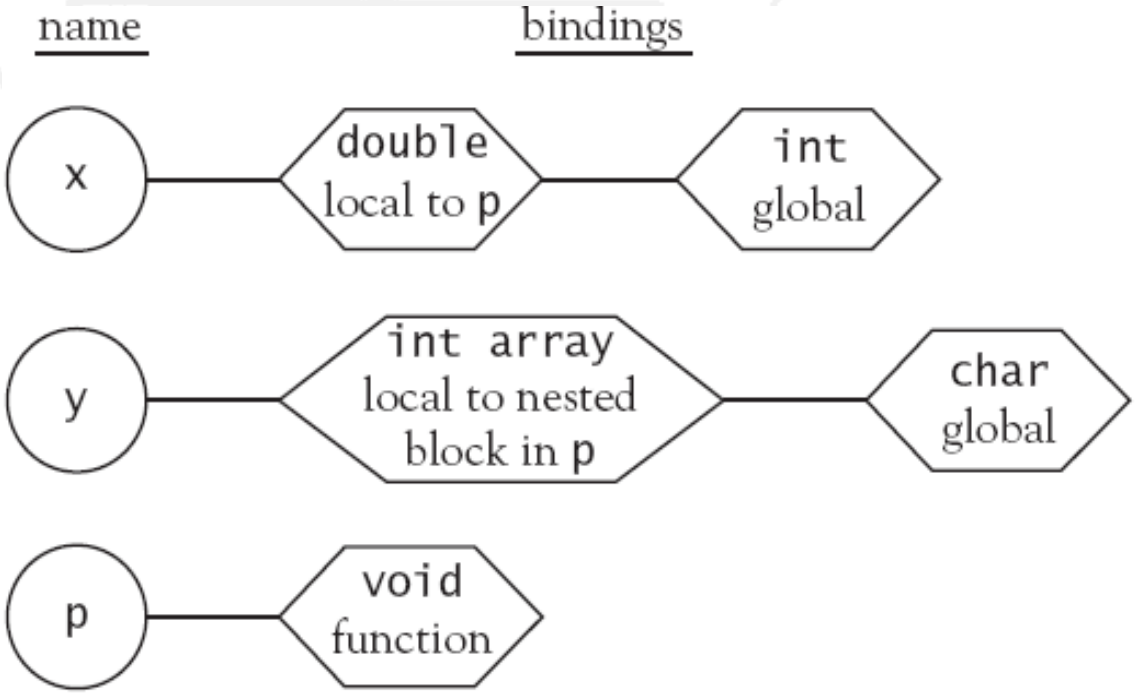
- The names in this program are x, y, p, q, and main, but x and y are each associated with three different declarations with different scopes.
- Right after the processing of the variable declaration at the beginning of the body of p the symbol table can be represented as in the figure below. (Since all function definitions are global in C, we do not indicate this explicitly in the attributes of the figure and subsequent figures.)



Symbol Table Structure

```
int x;
char y;
void p(){
    double x;
    ...
    { int y[10];
    ...
    }
    ...
}
void q(){
    int y;
    ...
}
main(){
    char x;
    ...
}
```

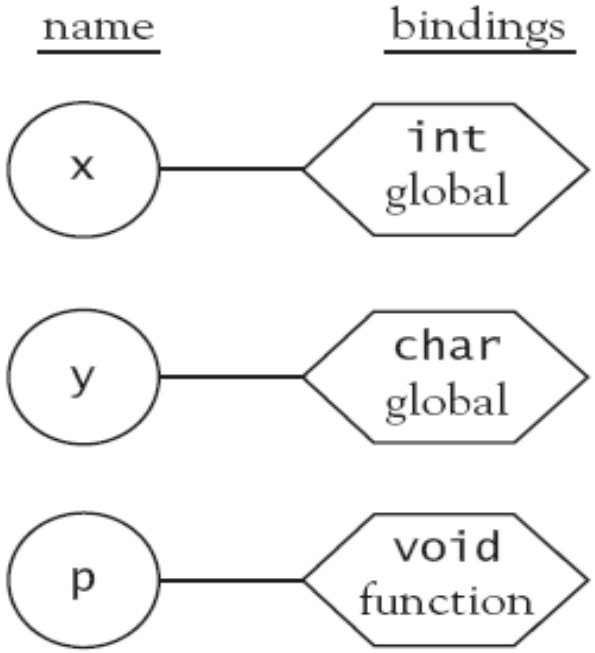
- After the processing of the declaration of the nested block in p, with the local declaration of y the symbol table is as shown in Figure below.



Symbol Table Structure

```
int x;
char y;
void p(){
      double x;
      ...
      { int y[10];
      ...
      }
      ...
}
void q(){
      int y;
      ...
}
main(){
      char x;
      ...
}
```

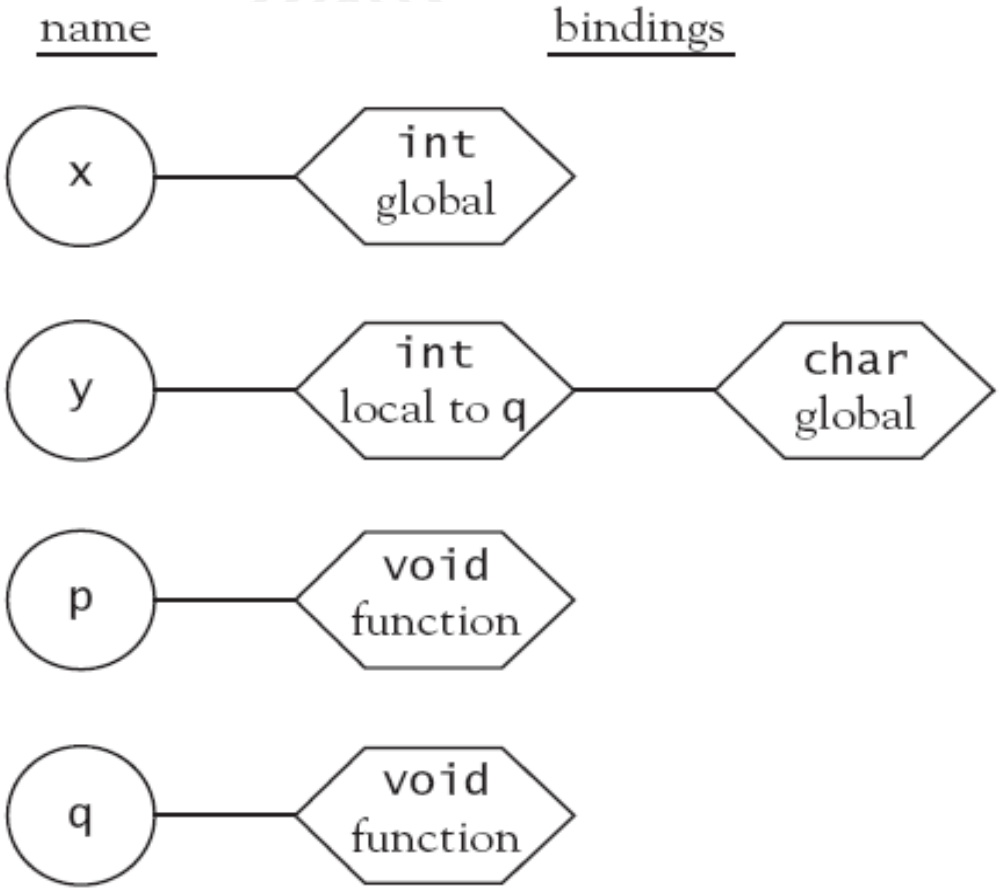
- When the processing of p has finished, the symbol table becomes as shown in the figure below (with the local declaration of y popped from the stack of y declarations and then the local declaration of x popped from the x stack).



Symbol Table Structure

```
int x;
char y;
void p(){
      double x;
      ...
      { int y[10];
      ...
      }
      ...
}
void q(){
      int y;
      ...
}
main(){
      char x;
      ...
}
```

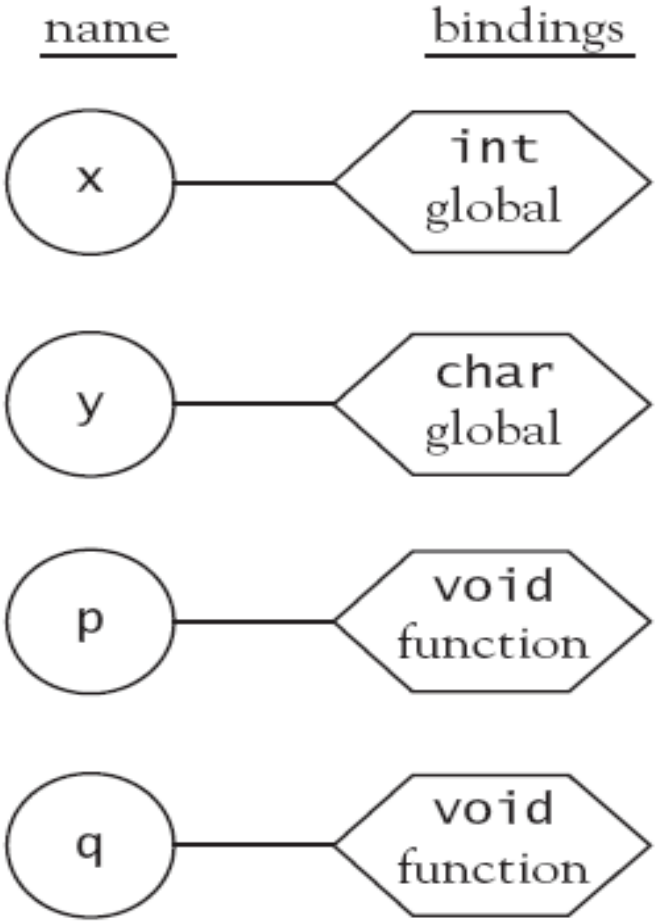
- After q is entered, the symbol table becomes like the one shown in the figure below.



Symbol Table Structure

```
int x;
char y;
void p(){
      double x;
      ...
      { int y[10];
      ...
      }
      ...
}
void q(){
      int y;
      ...
}
main(){
      char x;
      ...
}
```

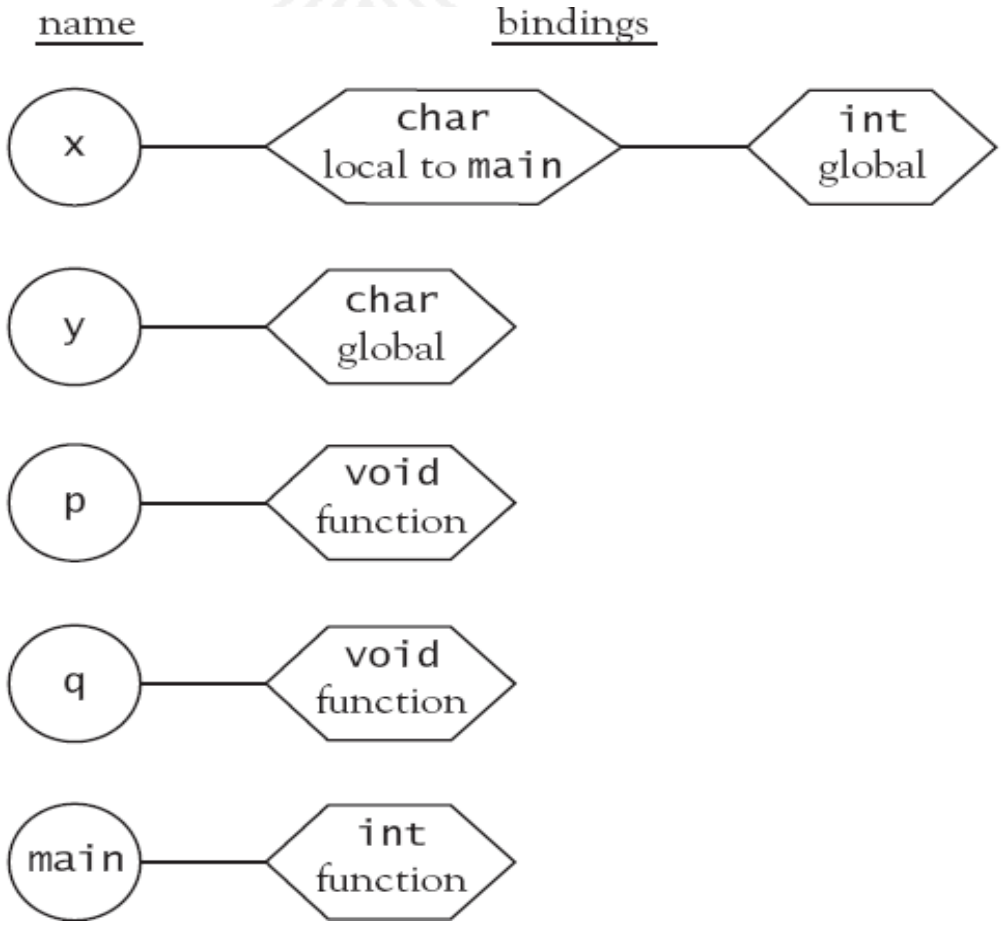
- When q exits, the symbol table is as shown in figure



Symbol Table Structure

```
int x;
char y;
void p(){
      double x;
      ...
      { int y[10];
      ...
      }
      ...
}
void q(){
      int y;
      ...
}
main(){
      char x;
      ...
}
```

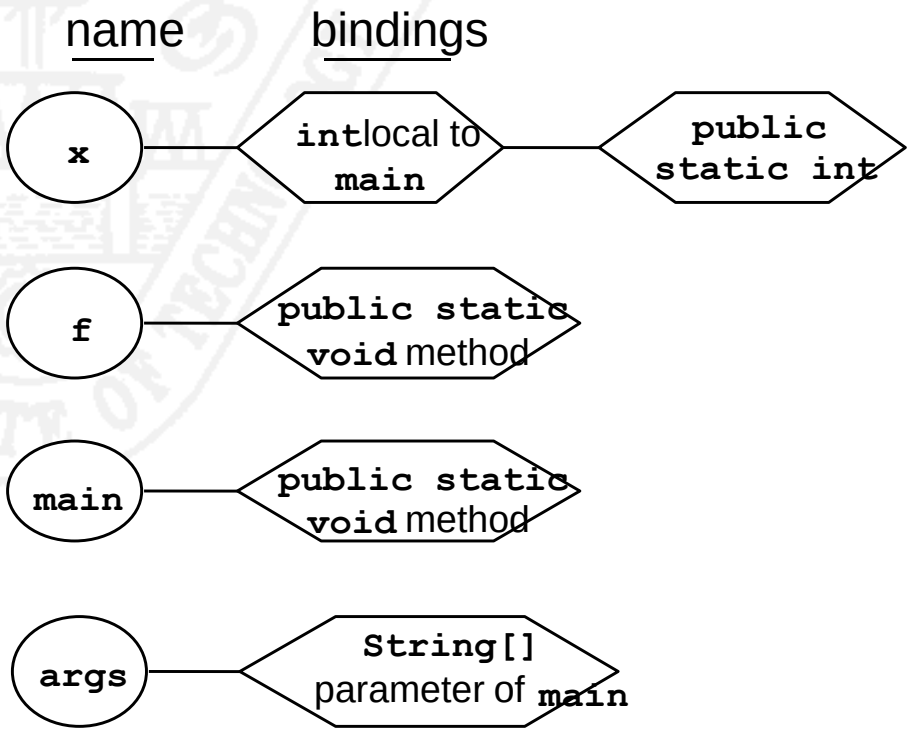
- Finally, after main is entered, the symbol table is as shown in the figure below.



Symbol table structure

- Stacks of declarations under each name. For example the table for the `Scope` class of slide 41 would look as follows inside `main` (using lexical scope):

```
public class Scope
{ public static int x = 2;
  public static void f()
  { System.out.println(x); }
  public static void main(String[] args)
  { int x = 3;
    f();
  }
}
```



Symbol table structure

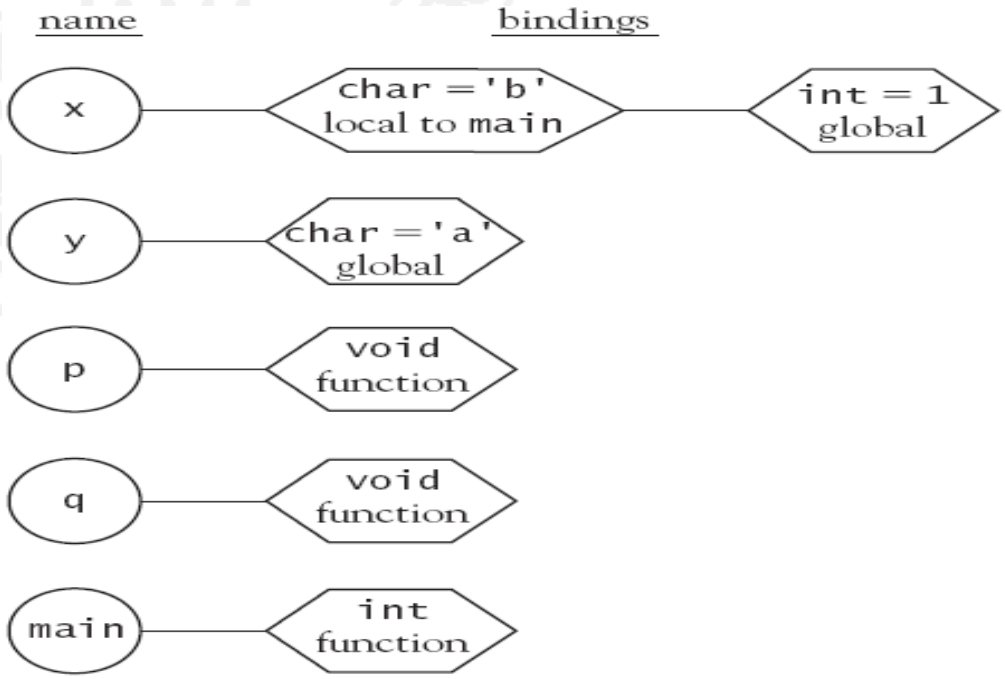
```
(1) #include <stdio.h>

(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) }
```

- If the symbol table were constructed dynamically as execution proceeds.
- First, execution begins with main, and all the global declarations must be processed before main begins execution, since main must know about all the declarations that occur before it. Thus, the symbol table at the beginning of the execution is as shown below….



Symbol table structure

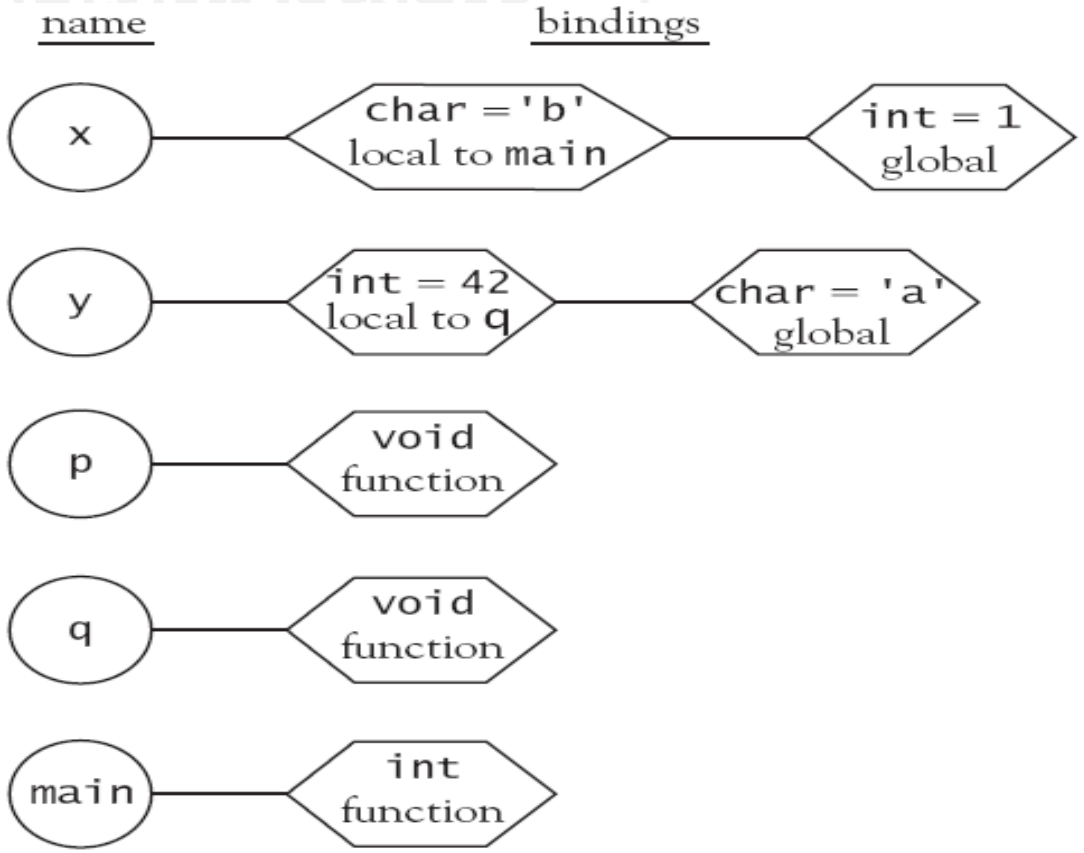
```
(1) #include <stdio.h>

(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) }
```

- Now main proceeds to call q, and we begin processing the body of q.
- The symbol table on entry into q is then as shown in the figure below.



Symbol table structure

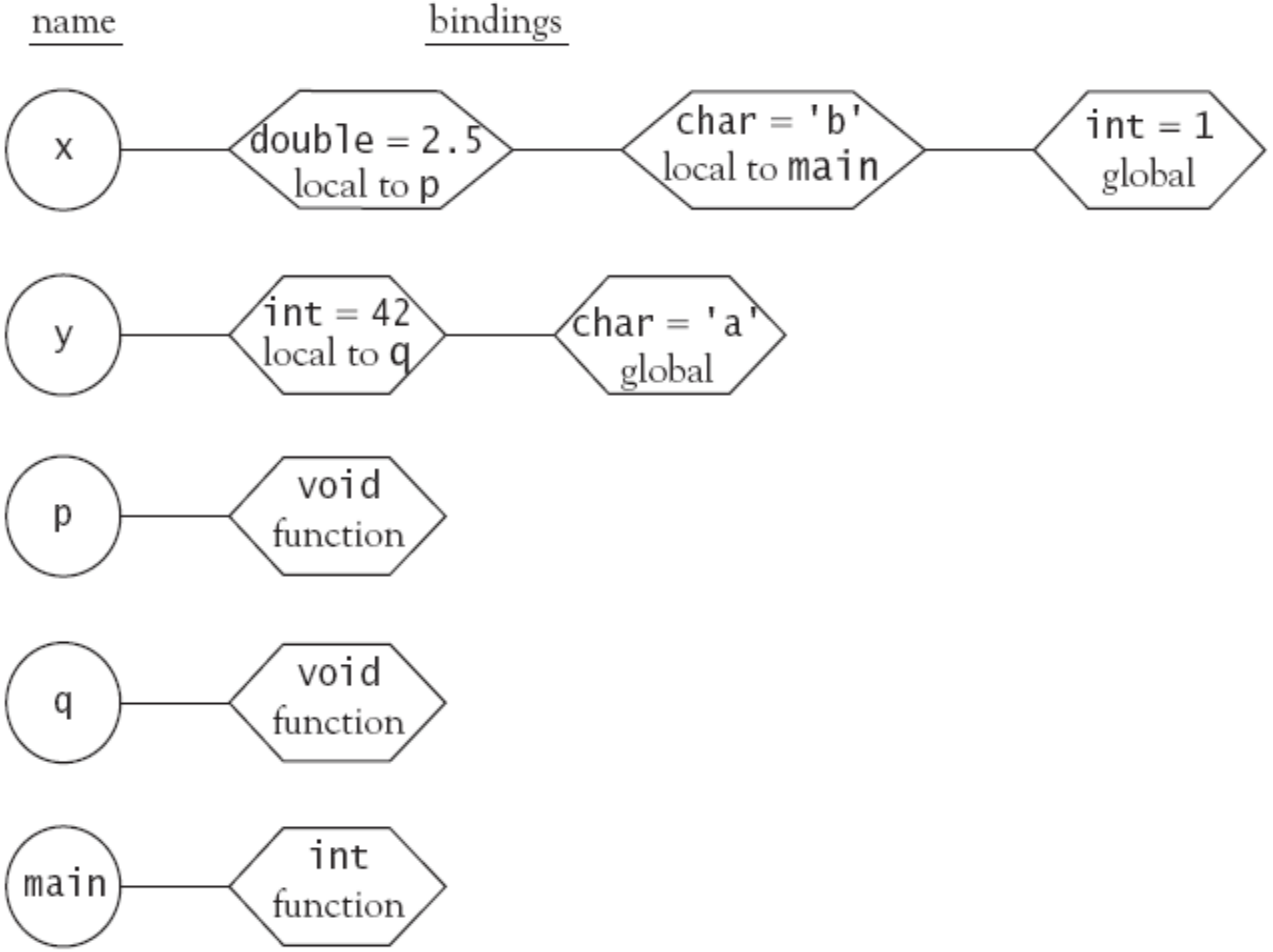
```
(1) #include <stdio.h>

(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) }
```

- q now calls p, and the symbol table becomes like the one shown in the figure below



Symbol table structure

```
(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) }
```

- What is the output of the program shown?
- the actual output of this program (using lexical scoping, which is the standard for most languages, including C) is:

  1

  a
- However, using dynamic scoping, the nonlocal references to y and x inside p and q, respectively, can change, depending on the execution path.
- In this program, the printf statement at line 12 is reached (with a new declaration of x as the character “b” inside main).
- Thus, the printf statement will print this character interpreted as an integer (because of the format string "%d\n") as the ASCII value 98.
- Second, the printf reference to y inside p (line 6) is now the integer y with value 42 defined inside q,
- This value will now be interpreted as a character (ASCII 42 = ‘*’), and the program will print the following:

  98

  *

Symbol table structure

```
(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) }
```

- Single table for an entire program, with insertions on entry into a scope and deletions on exit- is appropriate only for the simplest languages, such as C and Pascal.
- Each of the two struct declarations in this code (lines 1–5 and 7–11) must contain the further declarations of the data fields within each struct.
- These declarations must be accessible (using “dot”) whenever the struct variables themselves (x and y) are in scope.
- This means two things:

(1) A struct declaration actually contains a local symbol table itself as an attribute (which contains the member declarations)

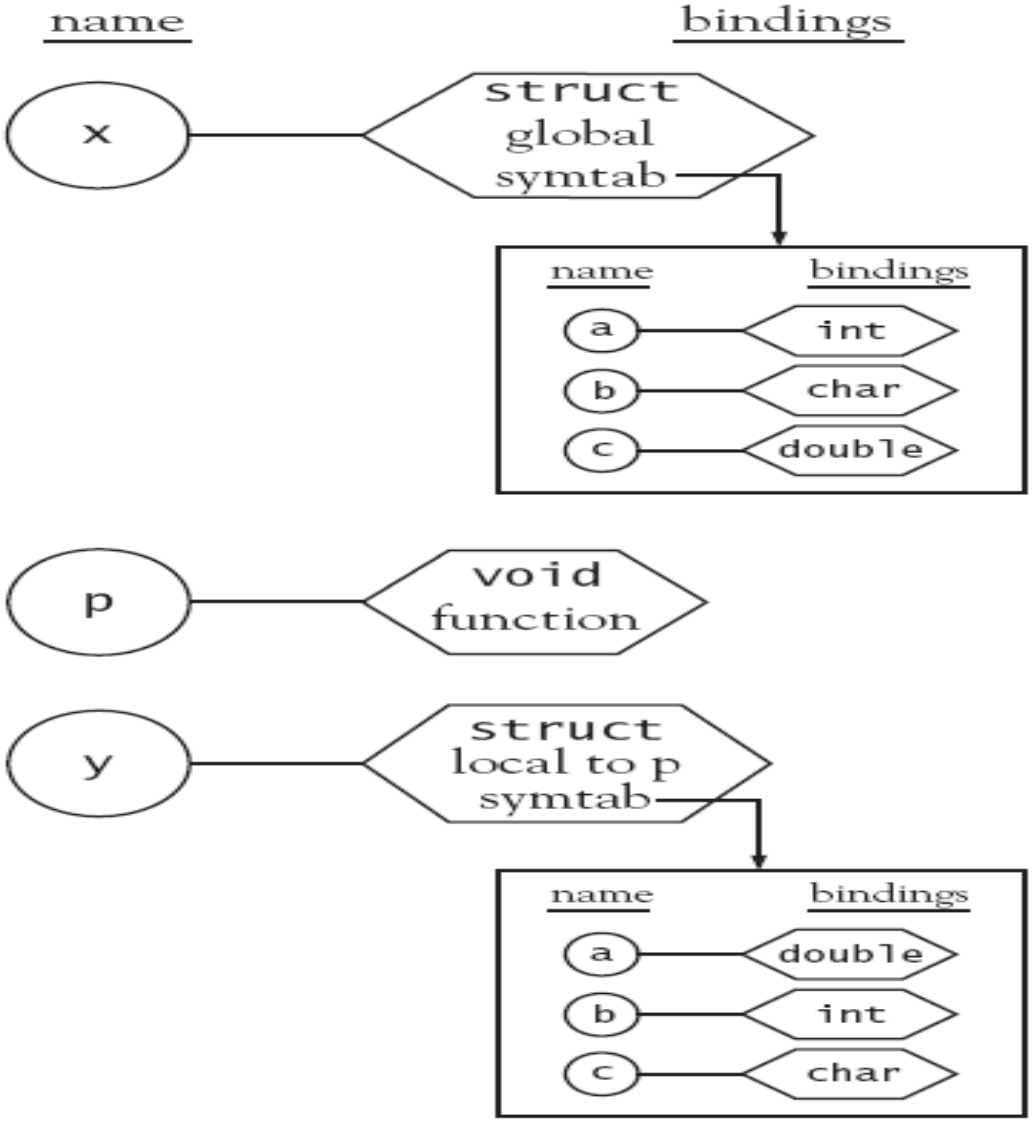
(2) this local symbol table cannot be deleted until the struct variable itself is deleted from the “global” symbol table of the program.

Symbol table structure

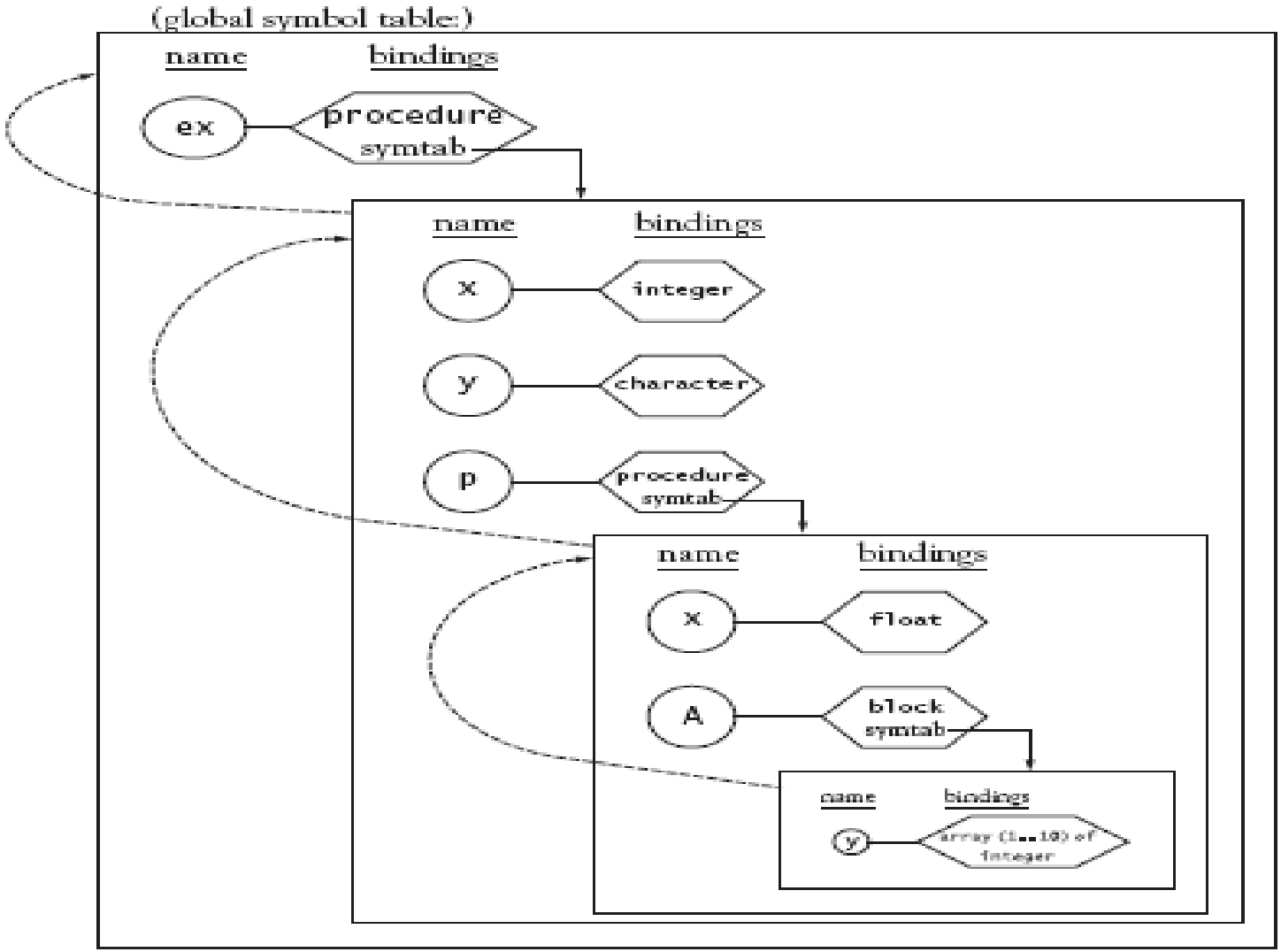
```
(1)  struct{
(2)      int a;
(3)      char b;
(4)      double c;
(5)  } x = {1, 'a',2.5};

(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) }
```



Symbol table structure



Symbol table structure of Ada Code

Name resolution and Overloading

- An important question about declarations and the operation of the symbol table in a programming language is to what extent the same name can be used to refer to different things in a program.

  ‘+’ operator

- The addition operation denoted by a + sign, which is typically a built-in binary infix operator in most languages and whose result is the sum of its two operands.
- This simple + sign refers to at least two (and often more) completely different operations: integer addition and floating-point addition
- Denoted in assembly language by two different symbols ADDI and ADDF
- The + operator is then said to be **overloaded**.

Name resolution and Overloading

- It would be a major annoyance if a programming language required programmers to use two different symbols for these operations (say +% for integer addition and +# for floating-point addition).
- How does a translator disambiguate (distinguish between) these two uses of the “+” symbol?
- 2+3, we mean integer addition
- If we write 2.1+3.2, we mean floating-point addition.
- Of course, 2+3.2 is still potentially ambiguous, and the rules of a language must deal in some way with this case (most languages automatically convert 2 to 2.0, but Ada says this is an error).
- It’s a major advantage for a language to allow overloading of operators and function names based in some way on data types

Name resolution and Overloading

- The basic method for allowing overloaded functions is to expand the operation of look up based not only on the name of a function but also on number of its parameters and their data type.

- This process of choosing a unique function among many with the same name is called “name resolution” or “overload resolution”.

Overloaded max function in C++

```
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);
```

Which max?

- max(2,3); // calls max #1
- max(2.1,3.2); // calls max #2
- max(1,3,2); // calls max #3

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

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

or

max(2.1,3.0); // max #2

So it is illegal.

Name resolution and overloading

```
typedef struct A A;
struct A{
int data;
A * next;
};
```

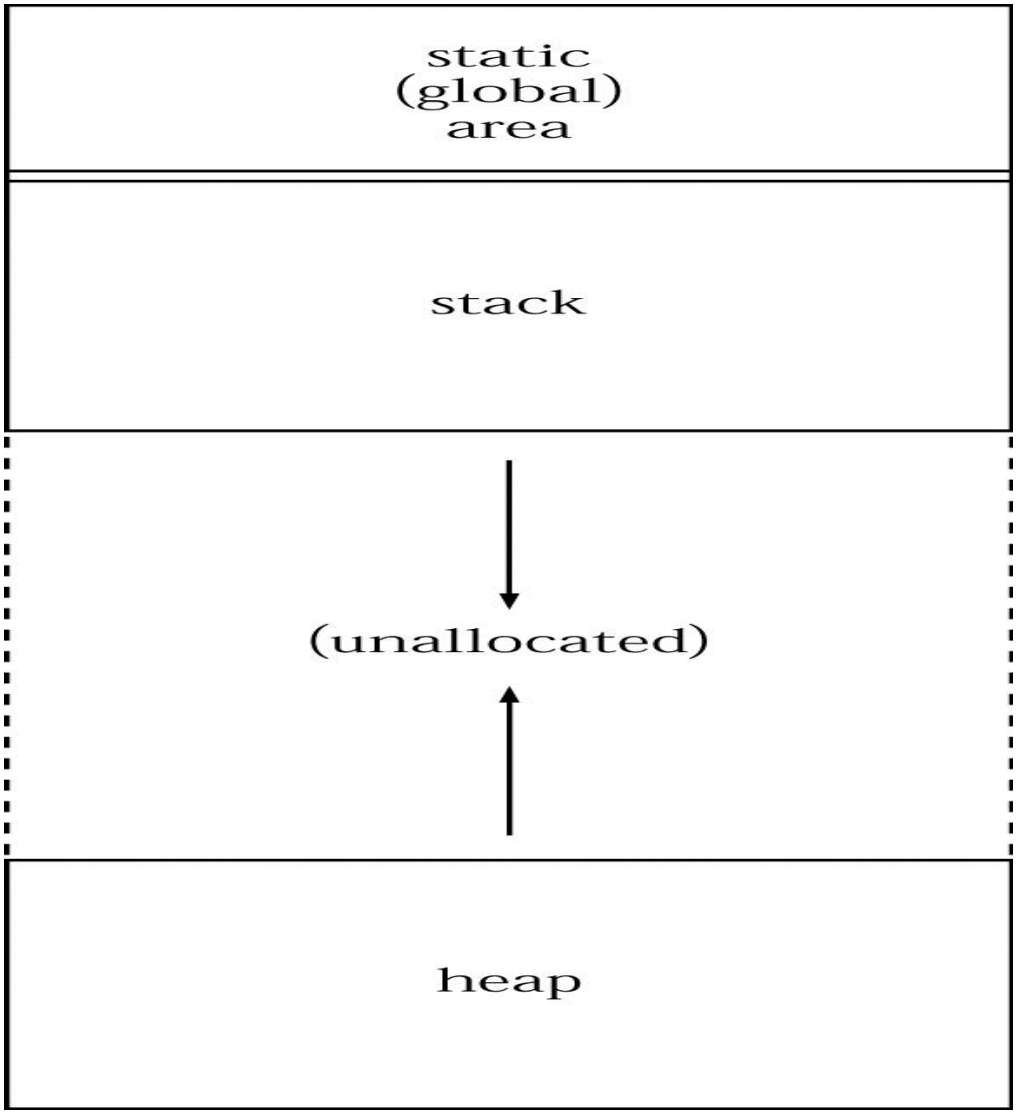
- Notice that we have used the same name, A, both for the struct name and the typedef name.
- This is legal in C, and it is useful in this case, since it reduces the number of names that we need to remember.
- This implies that struct names in C occupy a different symbol table than other names, so the principle of unique names for types (and variables) can be preserved.

Name resolution and overloading

- Some languages have extreme forms of this principle, with different symbol tables for each of the major kinds of definitions, for example: use the same name for a type, a function, and a variable.
- Java is a good example of this extreme form of name overloading, where the code below is perfectly acceptable to a Java compiler.

```
class A
{       A A(A A)
        { A:
                for(;;)
                { if (A.A(A) == A) break A; }
                return A;
        }
}
```

Structure of a typical environment



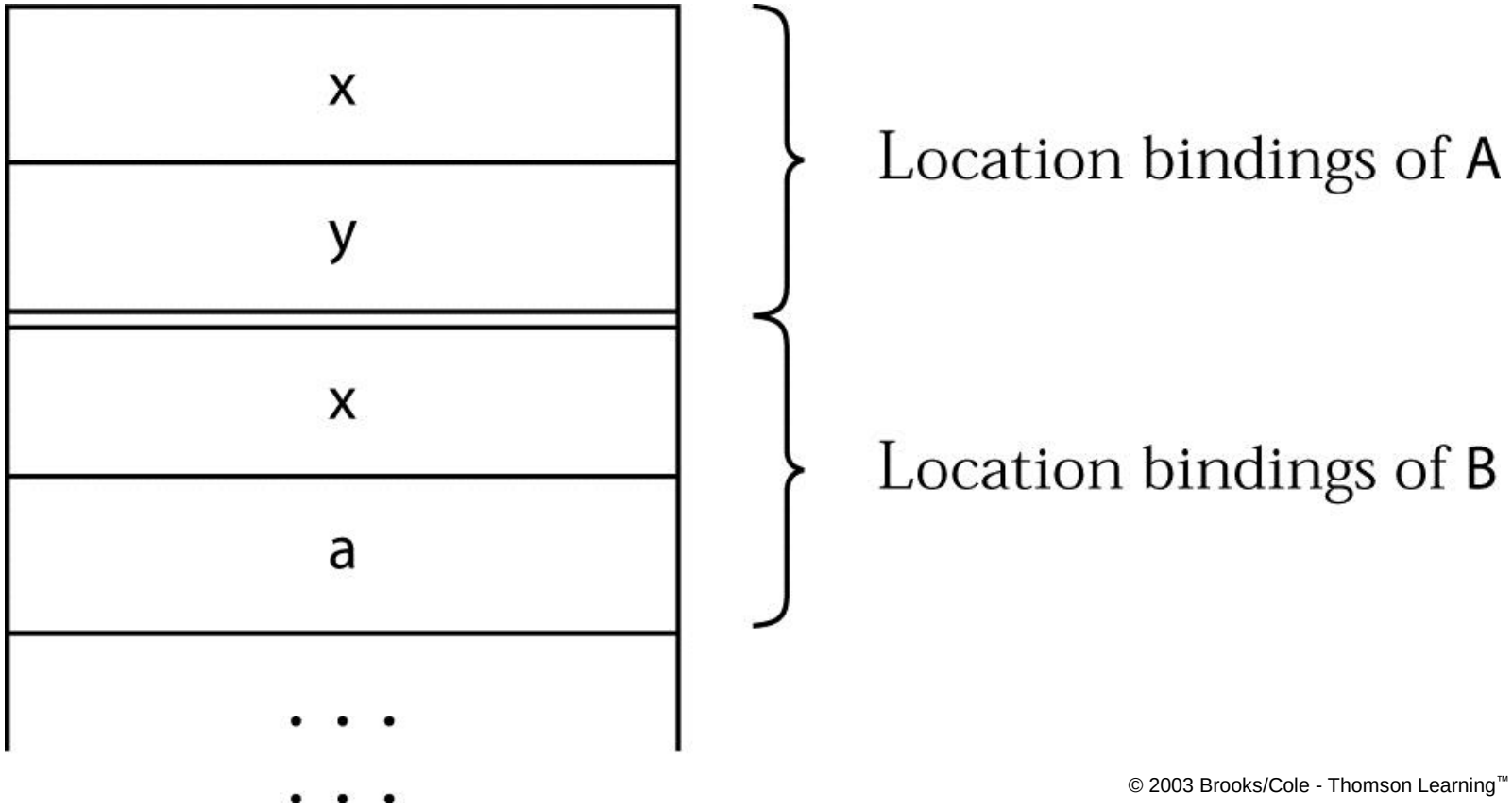
Example of stack-based allocation in C within a procedure:

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

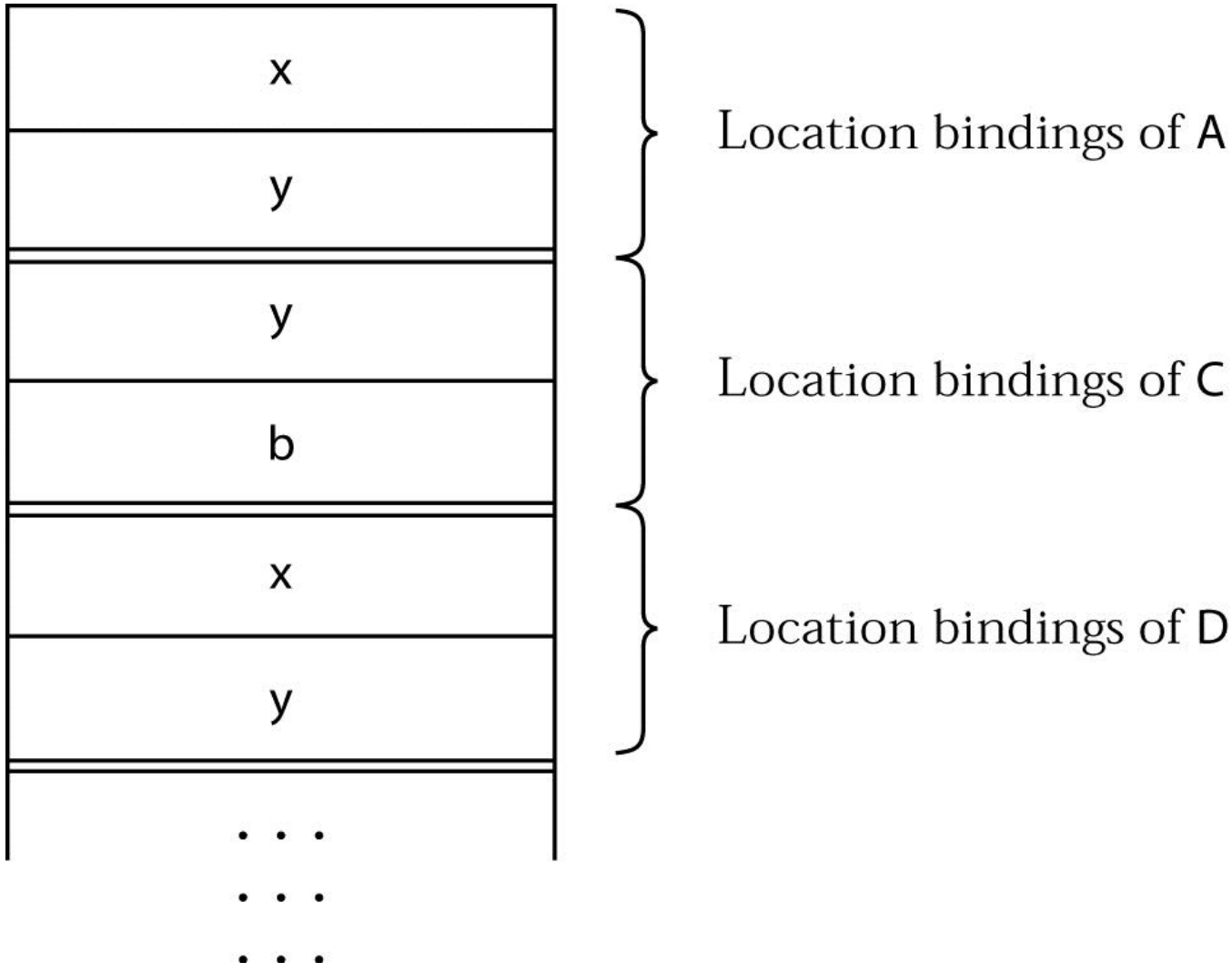
Point #1

Point #2

Stack at Point #1:



Stack at Point #2:



Variables and Constants

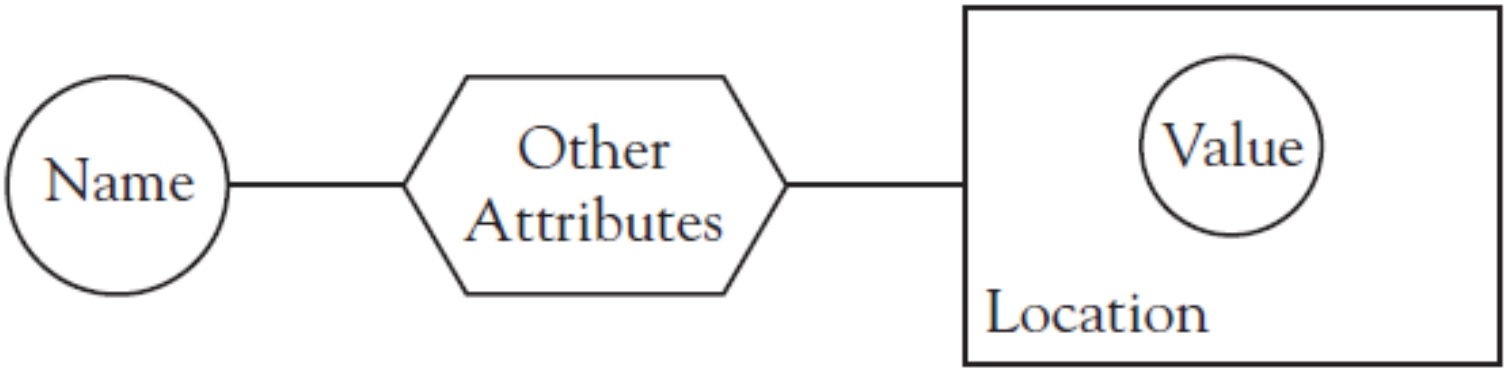
- Although references to variables and constants look the same in many programming languages, their roles and semantics in a program are very different.

➢ Objectives

- Clarifies and distinguishes their basic semantics.

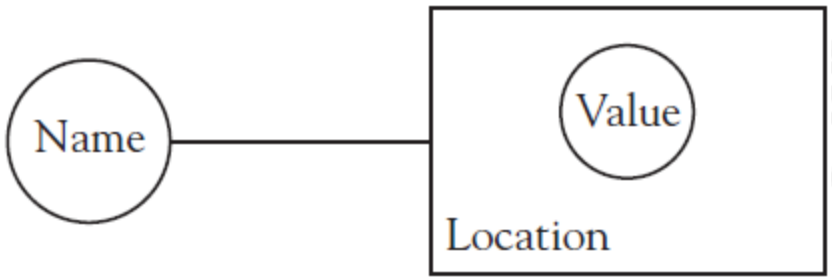
Variables and Constants

- A variable is an object whose stored value can change during execution.
- A variable can be thought of as being completely specified by its attributes, which include its name, its location, its value, and other attributes such as data type and size of memory storage.
- A schematic representation of a variable can be drawn as shown in Figure



Variables and Constants

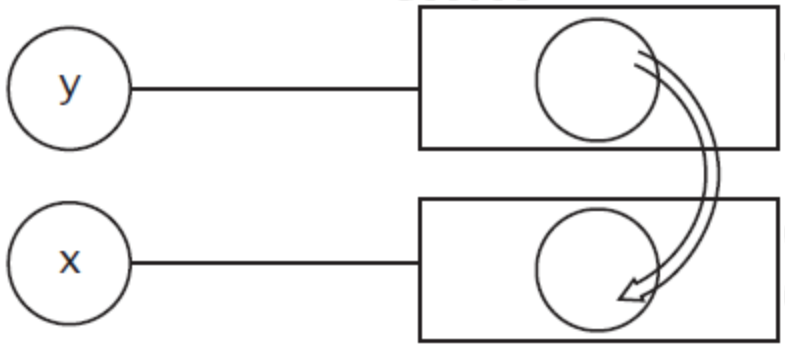
- Often we will want to concentrate on name, location, and value of a variable as being its principal attributes and then we picture a variable in the diagram shown below, which is known as a **box-and-circle diagram**.



- In a box-and-circle diagram, the line between the name and the location box represents the binding of the name to the location by the environment.
- The circle inside the box represents the value bound by the memory—that is, the value stored at that location.

Variables and Constants

x = y can be viewed as shown below, with the double arrow indicating copying.



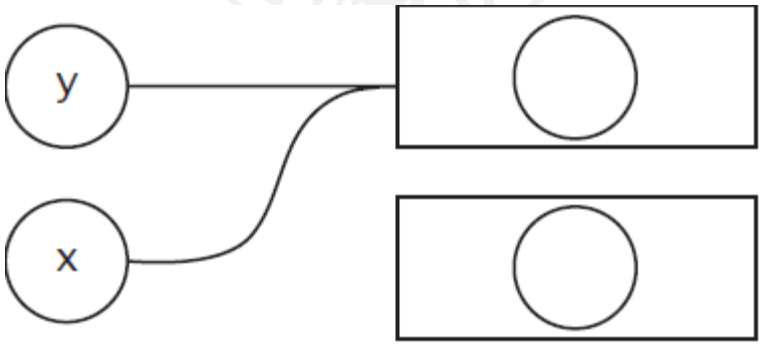
- Since a variable has both a location and a value stored at that location, it is important to distinguish clearly between the two.
- However, this distinction is obscured in the assignment statement, in which y on the right-hand side stands for the value of y and x on the left-hand side stands for the location of x.

Variables and Constants

- For this reason, the value stored in the location of a variable is sometimes called its **r-value** (for right-hand side value), while the location of a variable is its **l-value** (for left-hand side value).

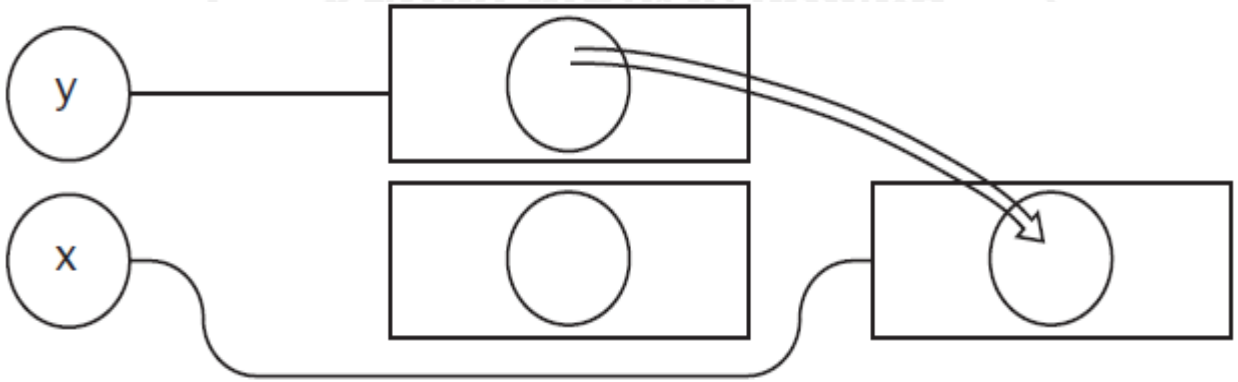
```
x + 1 = 2;    //is illegal (x + 1 is not an l-value), but
*(&x + 1) = 2;   //is legal.
```

- In some languages, assignment has a different meaning. Locations are copied instead of values.
- In this form of assignment, called **assignment by sharing**, case x = y has the result of binding the location of y to x instead of its value, as shown below



Variables and Constants

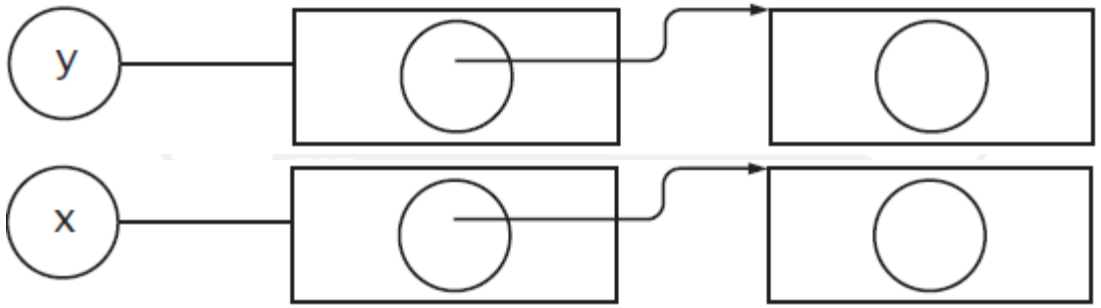
- An alternative, called **assignment by cloning**, is to allocate a new location, copy the value of y, and then bind x to the new location, as shown below



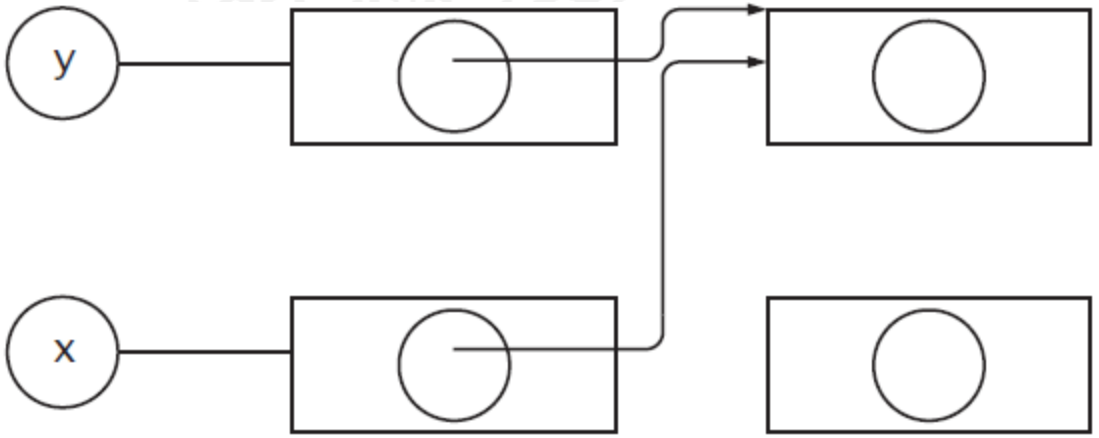
- In both cases this interpretation of assignment is sometimes referred to as **pointer semantics** or **reference semantics** to distinguish it from the more usual semantics, which is sometimes referred to as **storage semantics** or **value semantics**.

Variables and Constants

- Figure below shows **assignment by sharing**, with the name x being associated directly to a new location (that of y).

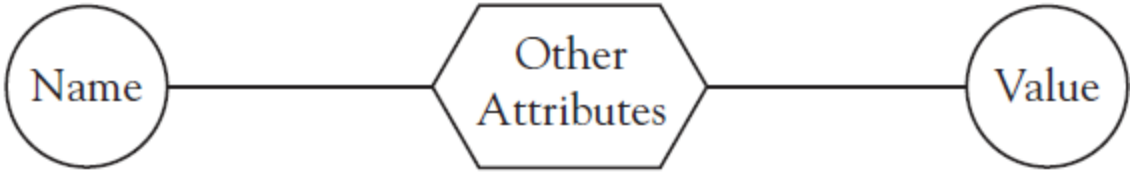


- Then the assignment x = y has the effect shown below. Note that this is the same as if we wrote x = y for pointer variables x and y in C.



Variables and Constants

- A **constant** is an object whose value does not change throughout its lifetime.



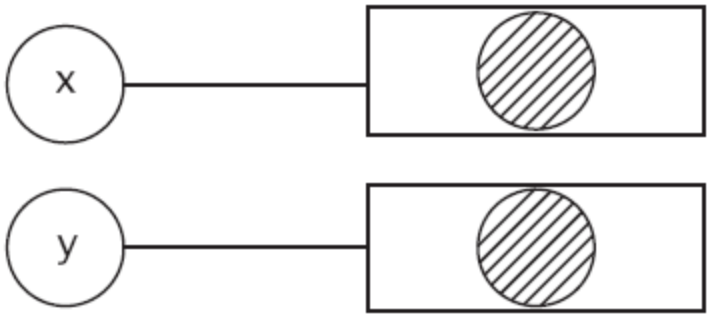
- Constants are often confused with literals: constants have names, literals do not.
- Constants may be: compile-time static (may not ever be allocated)
  load-time static and
  dynamic
- Compile-time constant in Java:
```
static final int zero = 0;
```
- Load-time constant in Java:
```
static final Date now = new Date();
```
- Dynamic constant in Java:
  any non-static final initialized in a constructor.

Aliases

- An alias occurs when the same object is bound to two different names at the same time. Example code in C

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

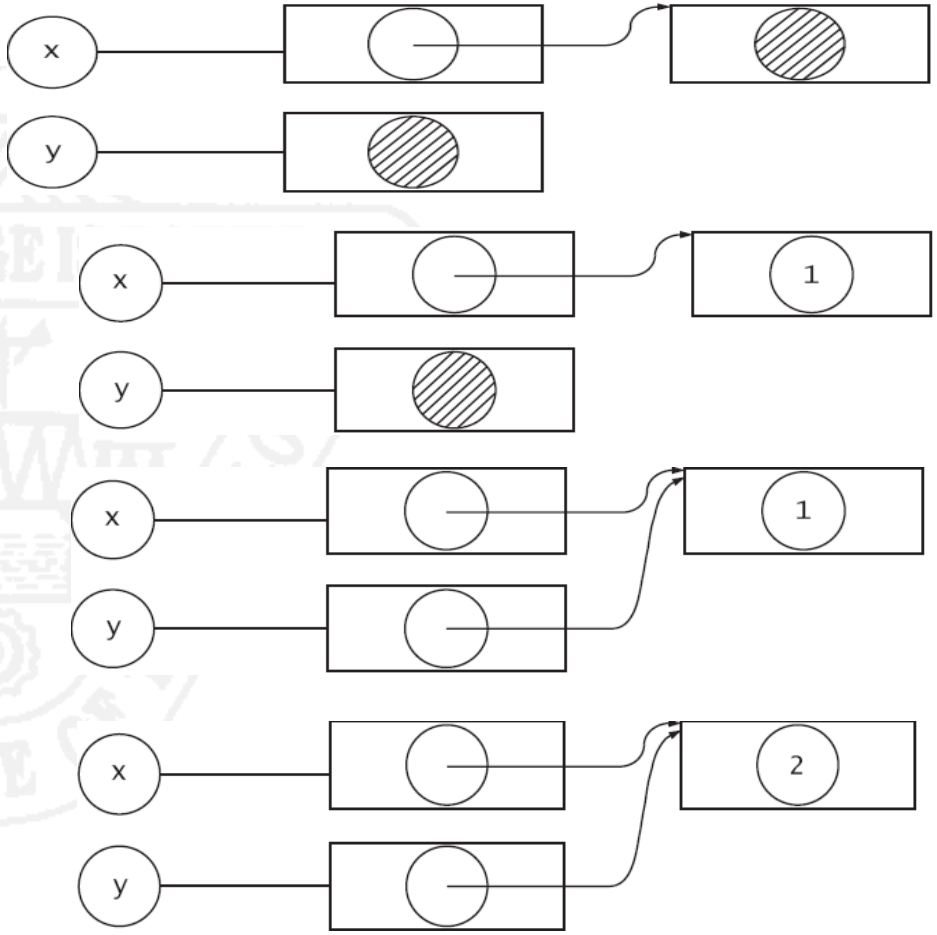
- After the assignment of x to y (line 4), *y and *x both refer to the same variable, and the preceding code prints 2.
- To see this clearly, record the effect of the preceding code using box-and-circle diagrams.
- After the declarations (line 1), both x and y have been allocated in the environment, but the values of both are undefined.



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

Aliases

- After line 2, *x has been allocated and x has been assigned a value equal to the location of *x, but *x is still undefined.
- After line 3, the situation is as shown



- Line 4 now copies the value of x to y, and so makes *y and *x aliases of each other. (Note that x and y are not aliases of each other.)
- Finally, line 5 results in the diagram as shown
- Aliases present a problem in that they cause potentially harmful **side effects.**

Aliases

- Aliases can be created is through assignment by sharing, since assignment by sharing implicitly uses pointers.
- Java is a major example of this kind of aliasing.

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

- Because of this aliasing problem in Java, Java has a mechanism for explicitly **cloning any object,** so that aliases are not created by assignment.
- For example, if we replace line 4 in the preceding code with:

```
int[] y = (int[]) x.clone();
```

then the value printed in line 6 is 1

Dangling References

- Dangling references are a second problem that can arise from the use of pointers.
- A dangling reference is a location that has been deallocated from the environment, but is still accessible within the program.
- In other words a dangling reference occurs if an object can be accessed beyond its lifetime in the environment.
- Dangling references are impossible in a garbage-collected environment with no direct access to addresses.
- A simple example of a dangling reference is a pointer that points to a deallocated object. In C, the use of the free procedure can cause a dangling reference, as follows:

```
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

- In C, it is also possible for dangling references to result from the automatic deallocation of local variables when the block of the local declaration is exited.
- This is because, as noted previously, C has the address of operator & that allows the location of any variable to be assigned to a pointer variable.
- Consider the following C code
- At line 5, when we exit the block in which y is declared, the variable x contains the location of y and the variable *x is an alias of y.

```
{ int * x;
  { int y;
    y = 2;
    x = &y;
  }
  /* *x is now a dangling reference */
}
```

- However, in the standard stack-based environment, y was deallocated on exiting from the block.
- A similar example is the following C code
- Whenever function dangle is called, it returns the location of its local automatic variable, which has just been deallocated.
- Thus, after any assignment such as y = dangle() the variable *y will be a dangling reference.

```
int * dangle(){
    int x;
    return &x;
}
```

Garbage

- One easy way to eliminate the dangling reference problem is to prevent any deallocation from the environment.
- This causes a problem called garbage.
- Garbage is memory that is still allocated in the environment but has become inaccessible to the program.
- Garbage can be a problem in a non-garbage collected environment, but is much less serious than dangling references.
- For this reason it is useful to remove the need to deallocate memory explicitly from the programmer (which, if done incorrectly, can cause dangling references), while at the same time automatically reclaiming garbage for further use.
- Language systems that automatically reclaim garbage are said to perform **garbage collection.**

References

Text book

- Kenneth C. Louden “Programming Languages Principles and Practice” second edition Thomson Brooks/Cole Publication.

Reference Books:

- Terrence W. Pratt, Masvin V. Zelkowitz “Programming Languages design and Implementation” Fourth Edition Pearson Education.
- Allen Tucker, Robert Noonan “Programming Languages Principles and Paradigms second edition Tata MC Graw –Hill Publication.