Names, Bindings, and Scopes

Lecture 09

Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu

MARSHALL

The Concept of Binding

- A ***binding*** is an ***association between an attribute and an entity***
  - such as between a variable and its type or value, or between an operation and a symbol.
- The time at which a binding takes place is called ***binding time***.
- ***Binding*** and ***binding times*** are prominent concepts in the semantics of programming languages.

The Concept of Binding

- Bindings can take place at *language design time*, *language implementation time*, *compile time*, *load time*, *link time*, or *run time*.
  - Example:
    - The asterisk symbol (*) is usually bound to the multiplication operation at *language design time*.
    - A data type, such as *int* in C, is bound to a range of possible values at *language implementation time*.
    - At *compile time*, a variable in a Java program is bound to a particular data type.
    - A variable may be bound to a storage cell when the program is *loaded into memory*.
    - A call to a library subprogram is bound to the subprogram code at *link time*.

The Concept of Binding

- Consider the following Java assignment statement:

```
count = count + 5;
```

- Some of the ***bindings*** and their ***binding times*** for the parts of this assignment statement are as follows:
  - The type of count is bound at *compile time*.
  - The set of possible values of count is bound at *language implementation time*.
  - The meaning of the operator symbol + is bound at *language design time*.
  - The internal representation of the literal 5 is bound at *compiler time*.
  - The value of count is bound at *execution time* with this statement.

Binding of Attributes to Variables

- A binding is ***static*** if it first occurs before run time begins and remains unchanged throughout program execution.

```
public class NewClass
{
    public static class superclass
    {
        static void print()
        {
            System.out.println("print in superclass.");
        }
    }
    public static class subclass extends superclass
    {
        static void print()
        {
            System.out.println("print in subclass.");
        }
    }

    public static void main(String[] args)
    {
        superclass A = new superclass();
        superclass B = new subclass();
        A.print();
        B.print();
    }
}
```

Binding of Attributes to Variables

- If the binding first occurs during run time or can change in the course of program execution, it is called ***dynamic***.

```
public class NewClass
{
    public static class superclass
    {
        void print()
        {
            System.out.println("print in superclass.");
        }
    }

    public static class subclass extends superclass
    {
        @Override
        void print()
        {
            System.out.println("print in subclass.");
        }
    }

    public static void main(String[] args)
    {
        superclass A = new superclass();
        superclass B = new subclass();
        A.print();
        B.print();
    }
}
```

Type Bindings

- Before a variable can be referenced in a program, it must be bound to a ***data type***.

- The two important aspects of this binding
  - How the type is specified
  - When the binding takes place

- Types can be specified statically through some form of ***explicit*** or ***implicit*** declaration.

Type Bindings: Static Type Binding

- An ***explicit declaration*** is a statement in a program that lists variable names and specifies that they are a particular type.
- An ***implicit declaration*** is a means of associating variables with types through ***default conventions***, rather than declaration statements.
  - In this case, the first appearance of a variable name in a program constitutes its implicit declaration.
- Both ***explicit*** and ***implicit*** declarations create ***static bindings*** to types.
- Most widely used programming languages that use static type binding exclusively and were designed since the mid-1960s require explicit declarations of all variables.

Type Bindings: Static Type Binding

- Implicit variable type binding is done by the ***language processor***, either a ***compiler*** or an ***interpreter***.
- There are several different bases for implicit variable type bindings.
- The simplest of these is ***naming conventions***.
  - In this case, the compiler or interpreter binds a variable to a type based on the ***syntactic form*** of the variable's name.
  - For example in Fortran
    - An identifier that appears in a program that is not explicitly declared is implicitly declared according to the following convention
      - If the identifier begins with one of the letters I, J, K, L, M, or N, or their lowercase versions, it is implicitly declared to be ***Integer*** type;
      - Otherwise, it is implicitly declared to be ***Real*** type.

Type Bindings: Static Type Binding

- Another kind of implicit type declarations uses ***context***.
- This is sometimes called ***type inference***.
- In the simpler case, the context is the ***type of the value*** assigned to the variable in a declaration statement.
- For example, in C# a ***var*** declaration of a variable must include an initial value, whose type is made the type of the variable.
- Consider the following declarations:

```
var sum = 0;
var total = 0.0;
var name = "Fred";
```

Type Bindings: Dynamic Type Binding

- With ***dynamic type binding***, the type of a variable is not specified by a declaration statement, nor can it be determined by the spelling of its name.
- Instead, the variable is bound to a type when it is assigned a value in an assignment statement.
- When the assignment statement is executed, the variable being assigned is bound to the type of the value of the expression on the right side of the assignment.
- Such an assignment may also bind the variable to an address and a memory cell, because different type values may require different amounts of storage.
- Any variable can be assigned any type value.
- Furthermore, a variable's type can change any number of times during program execution.

Type Bindings: Dynamic Type Binding

```
/* assign a string to variable x */
x = "The answer to all questions is ";
print(x);
/* now assign an integer to x */
x = 47;
print(x, ".\n");
```

Output:

```
The answer to all questions is 47.
```

Type Bindings: Dynamic Type Binding

- In Python, Ruby, JavaScript, and PHP, ***type binding is dynamic***.
- For example, a JavaScript script may contain the following statement:

```
list = [10.2, 3.5];
```

  - Regardless of the previous type of the variable named ***list***, this assignment causes it to become the name of a single-dimensioned array of length 2.
- If the statement

```
list = 47;
```

  - followed the previous example assignment, ***list*** would become the name of a scalar variable.

Scope

- The ***scope*** of a variable is the ***range of statements*** in which the variable is ***visible***.
- A variable is ***visible*** in a statement if it can be ***referenced*** in that statement.
- The ***scope rules*** of a language determine how a particular occurrence of a name is associated with a variable.
- In particular, ***scope rules*** determine how references to variables declared outside the currently executing ***subprogram*** or ***block*** are associated with their declarations and thus their attributes.

Scope

- A variable is ***local*** in a program unit or block if it is declared there.
- The ***nonlocal*** variables of a program unit or block are those that are ***visible*** within the program unit or block but are ***not declared*** there.
- ***Global*** variables are a special category of nonlocal variables.

Static Scope

- ALGOL 60 introduced the method of binding names to nonlocal variables called ***static scoping***, which has been copied by many subsequent imperative languages and many non-imperative languages as well.

- ***Static scoping*** is so named because the scope of a variable can be statically determined—that is, ***prior to execution***.

- This permits a human program reader (and a compiler) to determine the type of every variable in the program simply by examining its source code.

Static Scope

- When the reader of a program finds a reference to a variable, the attributes of the variable can be determined by finding the statement in which it is declared.

Static Scope

- In ***static-scoped languages*** with nested subprograms, this process can be thought of in the following way:
  - Suppose a reference is made to a variable ***x*** in subprogram ***sub1***.
  - The correct declaration is found by first searching the declarations of subprogram ***sub1***.
  - If no declaration is found for the variable there, the search continues in the declarations of the subprogram that declared subprogram ***sub1***, which is called its ***static parent***.
  - If a declaration of ***x*** is not found there, the search continues to the ***next-larger enclosing unit*** (the unit that declared sub1's parent), and so forth, until a declaration for ***x*** is found or the largest unit's declarations have been searched without success.
  - In that case, an ***undeclared variable error*** is reported.

Static Scope

- Consider the following JavaScript function, ***big***, in which the two functions ***sub1*** and ***sub2*** are ***nested***:

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

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

Static Scope

- Consider the following JavaScript function, ***big***, in which the two functions ***sub1*** and ***sub2*** are ***nested***:

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

- In some languages that use static scoping, regardless of whether nested subprograms are allowed, some variable declarations can be hidden from some other code segments.
- For example,
  - Consider again the function ***big***. The variable ***x*** is declared in both ***big*** and in ***sub1***, which is nested inside ***big***.
  - Within ***sub1***, every simple reference to ***x*** is to the local ***x***. Therefore, the outer ***x*** is hidden from ***sub1***.

Blocks

- Many languages allow new static scopes to be defined in the midst of executable code.
- This powerful concept, introduced in ALGOL 60, allows a section of code to have its own local variables whose scope is minimized.
- Such variables are typically stack dynamic, so their storage is allocated when the section is entered and deallocated when the section is exited.
- Such a section of code is called a ***block***.
  - Blocks provide the origin of the phrase block-structured language.

Blocks

- The C-based languages allow any ***compound statement*** (a statement sequence surrounded by matched braces) to have declarations and thereby define a new scope.
  - Such compound statements are called ***blocks***.
- For example, ***if*** list were an integer array, one could write

```
if (list[i] < list[j]) {
  int temp;
  temp = list[i];
  list[i] = list[j];
  list[j] = temp;
}
```

Blocks

- The ***scopes*** created by ***blocks***, which could be nested in larger blocks, are treated exactly like those created by subprograms.
  - References to variables in a block that are not declared there are connected to declarations by searching enclosing scopes (blocks or subprograms) in order of increasing size.
- Consider the following skeletal C function:

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

Declaration Order

- In C89, as well as in some other languages, all data declarations in a function except those in nested blocks must ***appear at the beginning of the function***.

- However, some languages—for example, C99, C++, Java, JavaScript, and C#—allow variable declarations to appear anywhere a statement can appear in a program unit.
  - However, in C99, C++, and Java, the scope of all local variables is from their declarations to the ends of the blocks in which those declarations appear.
  - In C#, the scope of any variable declared in a block is the whole block.

Scope and Lifetime

- Sometimes the ***scope*** and ***lifetime*** of a variable appear to be related.
- For example, consider a variable that is declared in a Java method that contains no method calls.
  - The ***scope*** of such a variable is from its declaration to the end of the method.
  - The ***lifetime*** of that variable is the period of time beginning when the method is entered and ending when execution of the method terminates.

```java
public static int minFunction(int n1, int n2) {
   int min;
   if (n1 > n2)
      min = n2;
   else
      min = n1;

   return min;
}
```

Scope and Lifetime

- This apparent relationship between ***scope*** and ***lifetime*** does not hold in other situations.
- In C and C++, for example, a variable that is declared in a function using the specifier ***static*** is statically bound to the scope of that function and is also statically bound to storage.
- So, its scope is static and local to the function, but its lifetime extends over the entire execution of the program of which it is a part.

```
#include<stdio.h>
int fun()
{
  static int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```