Concepts of programing Languages

Lecture 3-p2 :
Names, Binding, Type checking & Scopes.

Sudan University of Science and Technology

Dr. Aghabi Nabil Abosaif

20/10/2021

4.3 Storage Bindings and Lifetime

- The memory cell to which a variable is bound somehow must be taken from a pool of available memory. This process is called **allocation**.

- **Deallocation** is the process of placing a memory cell that has been unbound from a variable back into the pool of available memory.

- The **lifetime** of a variable is the time during which the variable is bound to a specific memory location.

- So, the lifetime of a variable **begins when it is bound to a specific** cell and **ends when it is unbound from that cell.**

storage bindings of variables

- To investigate storage bindings of variables, it is convenient to separate scalar (unstructured) variables into four categories, according to their lifetimes.
- These categories are
  1. Named Static,
  2. Stack-dynamic,
  3. Explicit Heap-dynamic,
  4. Implicit Heap-dynamic.

Static Variables

- **Static--bound** to memory cells before execution begins and remains bound to the same memory cell throughout execution, e.g., C and C++ static variables in functions.
- Also, when the **static modifier** appears in the declaration of a variable in a class definition in C++, Java, and C#, it also implies that the variable is a class variable, rather than an instance variable.
- **Advantages**: efficiency (direct addressing), history-sensitive subprogram support.
- **Disadvantage**:
  - lack of flexibility (no recursion)
  - storage cannot be shared among variables.

Stack-Dynamic Variables

- **Stack-dynamic**--Storage bindings are created for variables when **their declaration statements are elaborated.**

- **(A declaration is elaborated when the executable code associated with it is executed)**

- As their name indicates, stack-variables are allocated from run-time stack.
- In Java, C++, and C#, variables defined in methods by default stack dynamic.

- **Advantage**: allows recursion; conserves storage
- **Disadvantages**:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

Explicit Heap-Dynamic Variables

- They are **nameless** (abstract) **memory cells** that are ***allocated*** and ***de-allocated*** by explicit run-time instructions (which take effect during execution) written by the programmer.

- These variables, can only be referenced **through pointer or reference variables.**

- The heap is a **collection** of **s a hierarchical data structure** or **storage cells** whose organization is highly **disorganized** due to the **unpredictability** of its use.

As an example of explicit heap-dynamic variables, consider the following C++ code:

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

- Also **Java objects** are **explicitly heap dynamic** and are accessed through **reference variables**.
- Java has no way of explicitly destroying a heap-dynamic variable; rather, implicit garbage collection is used.

- **Advantage**: provides for dynamic storage management
- **Disadvantage**: inefficient and unreliable

Implicit heap-dynamic variables

- They are bound to heap storage ***only when*** they are assigned values.
- For example, consider the following statement in JavaScript:
  - highs = [74, 84, 86, 90, 71];
- Regardless of whether the variable named highs was previously used in the program or what it was used for, **it is now an array of five numeric values.**
- **Advantage**: flexibility (generic code)
- **Disadvantages**:
- Inefficient, because all attributes are dynamic
- Loss of error detection

```
int x; /* static stack storage */
void main() {
    int y; /* dynamic stack storage */
    char str; /* dynamic stack storage */
    str = malloc(50); /* allocates 50 bytes of
                dynamic heap storage */
    size = calcSize(10); /* dynamic heap storage */
    .
    .
    .
}
```

5. 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** or **assigned in that statement.**

- 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 .

- 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.

- Scoping can be of classes, packages, and namespaces

5.1 Static scoping

- The scope of a variable can be **statically** determined—that is, **prior to execution**.
- **Static scoping** is sometimes called **lexical scoping.**

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

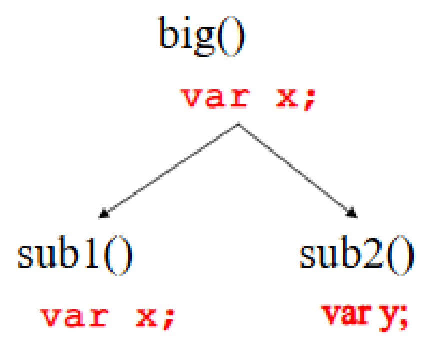
- There are **two categories of static-scoped** languages:
  - A **subprograms can be nested**, which creates nested static scopes.
  - A subprograms **cannot be nested**.

Static scoping(Cons.)

- **Nested scopes** are created only by **nested class** definitions and **blocks**.

- Ada, JavaScript, Common Lisp, Scheme, Fortran 2003+, F#, and Python **allow nested subprograms**, but the **C-based languages do not.**

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



Nested scopes Example

- **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 scoping(Cons.)

- 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 JavaScript 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**.

- Hidden variables can be accessed in some languages
  - E.g., Ada
    - big.x

5.2 Global Scope

- Some languages; C, C++, PHP, JavaScript, and Python, allow a program structure that is a sequence of function definitions, in which **variable definitions can appear outside** the **functions**.

- Definitions outside functions in a file create **global variables**, which potentially can be **visible to those functions**.

- C and C++ have both **declarations and definitions** of global data.
  - **Declarations** specify **types** and other attributes **but do not** cause **allocation** of storage.
  - **Definitions** specify **attributes** and cause **storage allocation**.

Global Scope(Cons.)

- A **C** program can have **any number** of compatible declarations, but only a single definition.
- A declaration of a variable **outside function definitions** specifies that the **variable is defined in a different file**.
- A global variable in **C** is implicitly visible in all subsequent functions in the file, **except those that include** a declaration of a local variable with the same name.
- A **global variable** that is defined after a function can be made visible in the function by declaring it to be **external**.

```
extern int  sum;
```

5.2 Dynamic Scope

- Dynamic scoping is based on the calling sequence of subprogram, not on their spatial relationship to each other.
- The **scope** can be determined only at run time.

- The scope of variables in APL, SNOBOL4, and the early versions of Lisp is dynamic.
- **Perl** and Common **Lisp** also allow variables to be declared to have dynamic scope

Dynamic Scope Example

- Consider the following two calling sequences:
- **big** calls **sub1**, **sub1** calls **sub2**
- big calls **sub2**

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

Dynamic Scope Example

- First, **`big`** calls **`sub1`**, which calls **`sub2`**.
  - In this case, the search proceeds from the local procedure, **`sub2`**, to its caller, **`sub1`**, where a declaration for **`x`** is found.
  - So, the reference to **`x`** in **`sub2`** in this case is to the **`x`** declared in **`sub1`**.
- Next, **`sub2`** is called directly from **`big`**.
- In this case, the dynamic parent of **`sub2`** is **`big`**, and the reference is to the **`x`** declared in **`big`**.
- Note that if static scoping were used, in either calling sequence discussed, the reference to **`x`** in **`sub2`** would be to **`big`'s** **`x`**.

5.3 Referencing Environments

- The referencing environment of a statement is <u>the collection of all variables that are visible in the statement.</u>
- **In a static scoped** language is the variables declared in its **local scope** *plus* the collection of all variables of **its ancestor** scopes.

Referencing Environments ..

- **For dynamic scoped** language:
- A subprogram is **active** if its execution has begun but has not yet terminated.

- The reference environment in a dynamically scoped language is the **locally declared** variables, plus the **variables of all other subprograms that are currently active**.

Named Constants

- A name constant is variable that is bound to a value **only once**.
- Useful as aids to **readability** and program **reliability.**
- E.g.In Java,
  - **final int len=100;**
- Ada and C++ allow dynamic binding of values to named constants, in C++:
  - **const int result = 2* width +1 ;**

Benefits of Constants

parameterize a program

```
void example() {
  int[] intList = new int[100];
  String[] strList =  new  String[100];
  . . .
  for  (index = 0; index < 100; index++) {
     . . .
  }
  . . .
  for  (index = 0; index < 100; index++) {
     . . .
  }
  . . .
  average = sum / 100;
  . . .
}
```

Benefits of Constants

parameterize a program

```
void example() {
  final int  len = 100;
  int[] intList =  new int[len];
  String[] strList =  new  String[len];
  . . .
  for (index = 0; index < len; index++) {
    . . .
  }
  . . .
  for (index = 0; index < len; index++) {
    . . .
  }
  . . .
  average = sum / len;
  . . .
}
```

Thank You
For Your Attention