Names, Bindings, and Scopes

Principles of Programming Language

S.Venkatesan

Reference: Robert W. Sebesta, Concepts of Programming Languages, 10th Edition, Pearson

Introduction

- A programming language variable can be characterized by a collection of properties, the most important is type.

- The data type designing have different issues however, the scope and lifetime of the variables are important.

- Functional programming languages allow expression to be named but in imperative languages expressions are assigned to variables (they are like named constants for imperative).

Names -- Identifiers

- Design Issues
  - Are names case sensitive?
  - Are the special words of the language reserved words or keywords?

Name Forms

- It is a string of characters used to identify some entity in a program.
- Fortran 95+ - upto 31 characters
- C99 – no limitation for internal names but only first 63 are significant. 31 characters for external.
- Representation – underscore, dollar, etc.
- C-based languages – Rose, ROSE, rose <- affects readability.
- Java – ParseInt -> affects writability

Special Words

- Keyword
- Reserved word

- COBOL – has 300 reserved words.
- Explicit import will use the pre-define words otherwise it will not impact.

Variables

- It is more than names for memory location.
- Numeric memory addresses for data with names. <- better readability as well as write and maintain.
- Six tuple characterization (name, address, value, type, lifetime and scope).

Address

- It is the machine memory address.
- At different times associated with different addresses.
- l-value – address of a variable.

- Aliases –
  - Multiple variables that have the same address.
  - It has hindrance that if one changes others also affects.
  - In C and C++ is with the union type.
  - Pointers
  - Reference variable

Type

- int
- char

Value

- The value of a variable is the contents of the memory cell or cells associated with the variable.
- Abstract Cell Vs Physical Cell (each cell) -> memory cell means abstract memory cell.
- It is sometime called as r-value – to access r-value, the l-value must be determined.

Binding

- It 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 binding time.
- Binding at language implementation time (for example, * at language design time), compile time, load time, link time or run time.
- Static vs Dynamic binding

Static Type Binding

- Explicit declaration – is a statement in a program that lists variables names and specifies that they are a particular type.

- Implicit declaration – through default conventions rather than declaration statements. (for example in fortran if variable starts with any of I, J, K, L, M or N or their lowercase it is implicitly declared to be Integer type)

Dynamic Type Binding

- Based on the valued it is assigned in the assignment statement.
  - List = [10.2, 13.5]
  - List = 47
- The option of dynamic type binding was introduced in C#2010.
  - A variable can be declared to use dynamic type binding by including the dynamic reserved word in it declaration.
  - dynamic any

Storage Bindings and Lifetime

- The process allocation is allotment of memory cell.
- Deallocation is making the memory cell to available memory.
- Lifetime of a variable is the time during which the variable is bound to a specific memory location.

Static Variables

- Are those that are bound to memory cells before program execution begins and remain until program execution terminates.
- No overhead at runtime.
- Reduced flexibility.
- Static specifier in c , c++ or Java (it is class variable rather than an instance variable)

Stack-Dynamic Variables

- Those whose storage bindings are created when their declaration statements are elaborated, but whose types are statically bound.

- For example, the variable declarations that appear at the beginning of a Java method are elaborated when the method is called and the variables defined by those declarations are deallocated when the method completes its execution.

- If variables declared in functions.

Explicit Heap-Dynamic Variables

- Are nameless memory cells that are allocated and deallocated by explicit run-time instructions written by the programmer.
- These variables, which are allocated from and deallocated to the heap, can only be references through pointer or reference variable.
- It is disorganized because of the unpredictability of its use.
- Bound to type at compile time and storage at run time.

Implicit Heap-Dynamic Variables

- Only when they are assigned values.
- It has runtime overhead.

Scope

- It is the range of statements in which the variable is visible.
  - Local
  - Global

- Static Scope – before execution
  - Those is which subprograms can be nested, which created nested static scopes, and those in which subprograms cannot be nested.
  - Static scopes are also created by subprograms but nested scopes are created only by nested class definitions and blocks.

Blocks

- Have its own local variables whose scope is minimized.

- Is it stack dynamic?

Declaration Order

- Must appear at the beginning of the function (C89).

- However, c99, c++, Java, JavaScript and C# allow variable declarations to appear anywhere a statement can appear in a program unit.

Global Scope

- Variable declarations can appear outside the functions.

- Need to give “extern” in C99 for visibility of variable, if variable is declared after function

Dynamic Scoping

- It is based on the calling sequence of subprograms, not on their spatial relationship to each other. Thus, the scope can be determined at run time.

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

One way the correct meaning of x can be determined during execution is to begin the search with the local declarations. This is also the way the process begins with static scoping, but that is where the similarity between the two techniques ends. When the search of local declarations fails, the declarations of the dynamic parent, or calling function, are searched. If a declaration for x is not found there, the search continues in that function's dynamic parent, and so forth, until a declaration for x is found. If none is found in any dynamic ancestor, it is a run-time error.

Consider the two different call sequences for sub2 in the earlier 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.

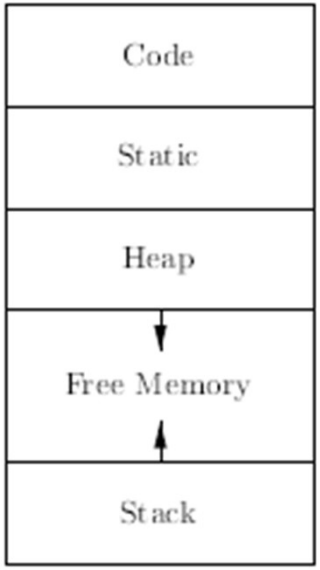
Note that if static scoping were used, in either calling sequence discussed, the reference to x in sub2 would be to big's x.

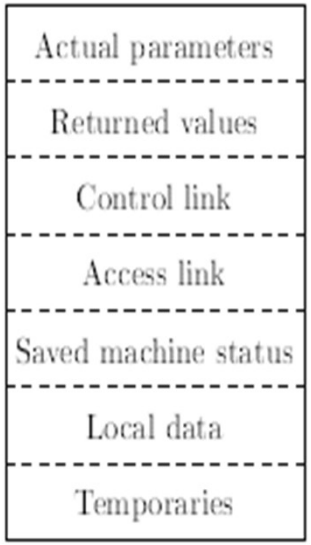
Scope and Lifetime

- Although scope and lifetime of the variable are clearly not the same, because static scope is textual, or spatial, concept whereas lifetime is a temporal concept, they at least appear to be related in this case.

- In C and C++, it does not hold for example, the static definer is statically bound to the scope of that function and is also statically bound to the 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.

Runtime Memory





Referencing Environments

- It is a collection of all variables that are visible in the statement.
- The referencing environment of a statement in a static-scoped language is the variables declared in its local scope plus the collection of all variables of its ancestor scopes that are visible.

Python Example

```
g = 3; # A global

def sub1():
    a = 5; # Creates a local
    b = 7; # Creates another local
    . . . <---------------------------- 1
  def  sub2():
  global g;  # Global g is now assignable here
    c = 9;  # Creates a new local
    . . . <---------------------------- 2
    def  sub3():
      nonlocal  c:  # Makes nonlocal c visible here
      g = 11;  # Creates a new local
      . . . <---------------------------- 3
```

The referencing environments of the indicated program points are as follows:

Point	*Referencing Environment*
1	local `a` and `b` (of `sub1`), global `g` for reference, but not for assignment
2	local `c` (of `sub2`), global `g` for both reference and for assignment
3	nonlocal `c` (of `sub2`), local `g` (of `sub3`)

```
void sub1() {
  int a, b;
  . . . <----------- 1
}  /* end of sub1 */
void sub2() {
  int b, c;
  . . . . <----------- 2
  sub1();
}  /* end of sub2 */
void main() {
  int c, d;
  . . . <----------- 3
  sub2();
}  /* end of main */
```

The referencing environments of the indicated program points are as follows:

Point	*Referencing Environment*
1	`a` and `b` of `sub1`, `c` of `sub2`, `d` of `main`, (`c` of `main` and `b` of `sub2` are hidden)
2	`b` and `c` of `sub2`, `d` of `main`, (`c` of `main` is hidden)
3	`c` and `d` of `main`

Named Constants

- It is a variable that is bound to a value only once.
- It increases the readability and program reliability for example pi.
- Another use of is parameterize.
- Ada and C++ allow dynamic binding of values to named constants.
  - C++ , const int result = 2 * width + 1
  - Java, final
  - C#, const and readonly
- Static (before run time - initialization) and dynamic at run time.