Variables - II

SCS 2207

Kokila Perera

Binding

Binding

Binding is the process of establishing an association such as

- Between an attribute and an entity
  - A variable and its type or value
- Between an operation and a symbol

Binding Times

The time which binding takes place

- **Language design time**
  - Ex: Bind operator symbols to operations
- **Language implementation time**
  - Ex: Bind floating point data type to a representation
- **Compile time:** When source code is converted into machine code
  - Ex: Bind a variable to a type in C or Java
- **Load time**
  - Relocatable machine code addresses are assigned to real addresses
  - Ex: Bind C static variables and FORTRAN 77 variables to memory cells
- **Runtime:** Program statements are executed at runtime
  - Ex: Bind a non static local variable to a memory cell

Binding Times

Example:

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

- The **type** of count is bound at **compile time**.
- The set of **possible values** of count is bound at **compiler design time**.
- The **meaning of the operator** symbol + is bound at **compile time**,
  - when the types of its operands have been determined.
- The **internal representation** of the literal 5 is bound at **compiler design time**.
- The **value** of count is bound at **execution time** with this statement.

Binding Attributes to a Variable

- **Name Binding**
  - Generally occurs at the compile time when compiler executes the declaration
- **Address Binding**
  - Global Variables - Load time
  - Local Variables - Runtime
- **Value binding**
  - Occurs at runtime – dynamic
- **Type binding**
  - Before a variable can be referenced in a program it must be bound to a data type.

Static and Dynamic Binding

Static Binding

- If binding first occurs *before run time* begins *and remains unchanged* throughout program execution, it is called static.

Dynamic Binding

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

Type Binding

Type Binding

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

The two important aspects of this binding are

- How the type is specified.
- When the binding takes place.

Static Type Binding

Static Type Binding

Type of variables are known at the time of compilation.

- By explicit or implicit declaration of variables.

Advantages:

- Certain errors in programs can be detected earlier.

Variable Declaration

Communicate to the translator information about the properties of the data objects needed during the program execution.

Specify types and other attributes

Both explicit and implicit declarations create static bindings to types

- Explicit declaration
  - Is a statement in a program that lists variable names and their type
- Implicit declaration
  - Variables are assigned types through default conventions based on the syntactic form of the variable's name.
  - The first appearance of a variable name in a program constitutes its implicit declaration

Variable Declaration: Initialization

A declaration may also specify the value of the data object

- Constant : a data object with a name that is bound to a value(s) permanently during its lifetime.
  - Java: `Integer.MAX_VALUE`
  - C: `INT_MAX`
- Literal (Literal constant) : a constant whose name is the written representation of its value.
  - Example : 21
  - The name of the literal is represented as the sequence of two characters “2” and “1”
  - The value of the literal 21 is represented as a sequence of bits in storage during program execution.

Example: Literals

Example:

```
…
int y;
…
y = 10;
….
```

During the program execution,

- The name of the literal will be represented by the character sequence “1” and “0”.
- Its value be stored in the memory as 10.

Implicit Variable Declaration

By naming conventions

- FORTRAN :
  - Support both explicit and implicit declaration
  - Names begin with I,J,K,L,M,N or lower case of these characters:
    - Are treated as Integer
    - Otherwise, Real

By special characters for the names of specific types to begin with

- Perl
  - $ - scalar variable (can store a string or a numeric value)
  - @ - array variable
  - % - hash structure

Implicit Variable Declaration

By type inference (using context)

- C#: **var** declaration of a variable must include an initial value, whose type is made the type of the variable

```
var sum = 20;
var total = 22.07;
var name = "SCS 2207";
```

Dynamic Type Binding

Dynamic Type Binding

A variable is bound to a type when it is assigned a value in an assignment statement.

- Type is not specified by a declaration statement or by the syntactic form of the name.

The type of a variable can change during the course of the program execution.

- Type should be re-determined on every assignment.
- Run time system must keep track of types of variables.

Dynamic Type Binding

The cost of implementing **dynamic attribute binding** is relatively high

- Type checking is done at run time
- Every variable must have a descriptor to maintain the current type
- Storage used for a variable may change at different places of the program

Languages with dynamic type binding for variables are often implemented using interpreters

- It is difficult to change dynamically the types of variables in machine code

Dynamic Type Binding

Example:Javascript

```
...

list = [10.2, 3.5];        // single-dimensioned array of length 2

...

list = 22                  //integer

...
```

- Regardless of the previous type of the variable, it changes when an assignment statement is executed

Dynamic Type Binding

In C# dynamic reserved word

```
dynamic num
```

- Class members, properties, method parameters, method return values, and local variables can all be declared dynamic.
- It is useful when data of unknown type come into a program from an external source.

Dynamic Type Binding

Advantages:

- Programs can be more flexible and compact.
- Generic programs and functions can be developed.
- Example
  - A program to process numeric data in a language that uses dynamic type binding can be written as a generic program
  - I.e. it is capable of dealing with data of any numeric type

Disadvantages

- High run-time overhead (dynamic type checking)
- Type incompatibility errors cannot be detected by the compiler.

Storage Bindings and Lifetime

Storage Binding

Storage binding : binding memory cells to a variable.

Allocation

- Binding a memory location to a variable.
- Typically storage required for a variable is taken from a pool of available memory

Deallocation

- Unbinding the allocated memory location from a variable
- Releasing the memory cells allocated to a variable to the pool of available memory

Variable Lifetime

Data objects can be created and destroyed during execution.

Lifetime : is the period of time which a variable is bound to a memory location.

- Lifetime begins when a variable is bound to a specific cell(s)
- Lifetime ends when the variable is unbound from that cell(s)

Runtime Memory Structure

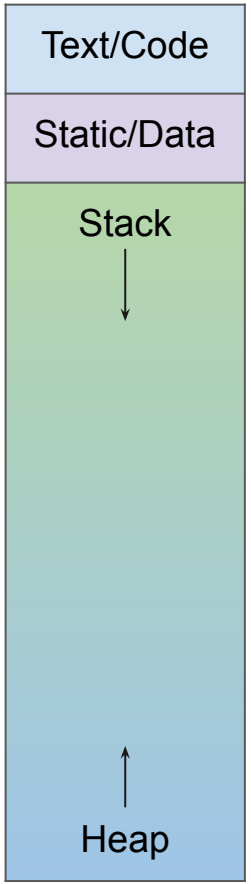
The logical organization of the memory used by a running program

3 main areas for variables:

- the global or static area
- the stack (contains "activation records")
- the heap (a highly disorganized collection of storage cells because of the unpredictability of its use)

The registers for temporary data

Read-only memory pages for constant data and code



Categories of Variables

Based on the variable lifetime

Static

Stack-dynamic

Explicit heap-dynamic

Implicit heap-dynamic

Static Variables

Bound to memory cells before program execution begins and remain bond to the same memory cell until program execution terminates

Used for global variables and local static variables in history sensitive subprograms

Static variables in a subprogram retain values between separate execution of the subprogram

- Examples:
  - In C and C++: static keyword allow to define a static variable

Static Variables

Advantages:

- Efficiency: Direct addressing
- Implementing history sensitive subprograms
- No runtime overhead in allocation and deallocation

Disadvantages

- Reduced flexibility: do no support recursion
- Storage cannot be shared among subprograms

Stack-dynamic Variables

Variables that are bound to memory cells when their declaration statements are **elaborated**

- I.e, binding takes place when execution reaches the code to which the declaration is attached

However the types are statically bound to stack dynamic variables.

Memory is allocated from run-time stack.

In Java, C++, and C#, variables defined in methods are stack dynamic.

Example:A variable declaration that appears at the beginning of a Java method

- The variable declaration is elaborated when the method is called,
- And deallocated when the method completes execution

Stack-dynamic Variables

- In some languages variable declarations are allowed to be stated anywhere in the code. (ex: C++, Java)
- In some implementations of such languages,
  - All stack dynamic variables declared in a function or a method are bound to the storage when the function/method begins to execute.
  - But the variable become visible only at the declaration

Stack-dynamic Variables

Advantages:

- Useful in recursive subprograms
  - each active call to subprogram can have its own version of local variables
- Conserve Storage

Disadvantages:

- Runtime overhead of allocation and deallocation (relative to static variables)
- Slower accesses because indirect addressing is required
- Not history sensitive

Explicit Heap-Dynamic Variables

Nameless (abstract) memory cells that are allocated and deallocated by explicit run-time instructions written by the programmer.

Allocated from and deallocated to the heap memory

Can only be referenced through pointer or reference variables.

- The pointer/reference is created similar to any other scalar variable.

Explicit Heap-Dynamic Variables

Explicit heap dynamic variables are created by

- An operator (ex: new operator in C++ and Java)
- A call to a system subprogram provided for that purpose (malloc, calloc, free in C)

Some languages include a subprogram or operator for explicitly destroying these heap dynamic variables..

Explicit Heap-Dynamic Variables

Example : Java

In Java all variables except primitive scalars are explicit heap dynamic.

```
Car testCar = new Car();
```

- New operator in Java creates a variable of a specific type and returns a pointer to that variable.
- Variable is bound to a type at compile time- the binding is static.
- Variables are bound to storage at run-time.
- Deallocation or unbinding of storage happens bt a implicit garbage collection.

Explicit Heap-Dynamic Variables

C++ example:

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

Explicit Heap-Dynamic Variables

C# has both explicit heap- dynamic and stack- dynamic objects, both of which are implicitly deallocated.

C# also supports C++-style pointers that are use to reference heap, stack, and even static variables and objects. The objects they reference on the heap are not implicitly deallocated.

Explicit Heap-Dynamic Variables

Advantage:

- To construct dynamic structures
  - Such as linked lists and trees, that need to grow and/or shrink during execution.
  - Such structures can be built conveniently using pointers or references and explicit heap-dynamic variables.

Disadvantage

- Difficulty of using pointer and reference variables correctly
- Cost of references to the variables,
- Complexity of the storage management implementation.

Implicit Heap-Dynamic Variables

Bound to heap storage only when they are assigned values.

Unlike explicit heap dynamic variables all the attributes of these variables are bound every time they are assigned (ex: name, type, address, value, etc)

When a value is re-assigned to the variable, allocation and deallocation occur

- Example

```
myarray = [74, 84, 86, 90, 71];
```

- Whether name myarray is previously used or not, this is now an array of 5 numeric values

Implicit Heap-Dynamic Variables

Advantage:

- High degree of flexibility: allows to write highly generic code

Disadvantage

- High run-time overhead: maintaining dynamic attributes
- Some errors cannot be identified by compilers

Languages such as APL, ALGOL 68, and LISP use implicit heap-dynamic variables.

Persistence

The lifetime of a variable or data extends beyond the execution of the program

- Data files are typically used for transferring persistent data into local program variables

Scope

Scope

The scope is the range of statements in a program over which it is visible.

- If it is visible it can be referenced in those statements
- If a variable is not visible to a statement the variable is said to be out of scope
- One of the basic reasons for scoping is **to keep variables with the same name in different parts of the program distinct from one another.**

Scope - Classification of variables based on scope

Local variables

- A variable is local in a program unit/block if it is declared there

Non-local variables

- Non local variable of a program unit/block are those are visible within the program or block but not declared there
- Global variables are a special type of nonlocal variables, that are visible from anywhere

Scope - Scope Rules

The **scope rules** of the language determine how a particular occurence of a name is associated with a variable.

And also they determine how references to variables declared outside the currently executing subprogram or block are associated with their declarations and thus their attributes.

```
function temp(x:real):real;
  begin
  return x * s;          //s is a free variable
 end;
```

Static Scope

The scope of the variable can be statically determined (prior to execution)

Also called lexical scope

Scope can be statically determined by examining the source code

- Enables the compiler as well as a human reader to determine the type of every variable in the program using the source code.

Static Scope

There are two categories of static scoped languages:

- Subprograms can be nested
  - Static scope is created by the subprograms, nested classes and blocks
  - Examples:
    - Ada, JavaScript, Common Lisp, Scheme, Fortran 2003+, F#, and Python

- Subprograms cannot be nested
  - Static scope will be created by the nested classes and blocks
  - Examples:
    - C based languages

Static Scope

If the scope of a variable inside a block has static scoping then

- the meaning of a block is unchanged when the name of the variable is changed everywhere inside the block.

Static Scope: Search Process

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

The correct declaration can be found out by first searching the declarations of the subprogram which we found it.

If no declarations found search continues in the declarations of the subprogram which declared (or enclosing) the first subprogram, which is called the **static parent**.

And until the declaration found search continues to the **static ancestors.**

Static Scope

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

Static Scope

In some languages, some variable declarations can be hidden from some other code segments

Variables can be hidden from a unit by having a "closer" variable with the same name

In Ada, hidden variables from ancestor scopes can be accessed with selective references, which include ancestors scope name.

Ex: big.x

Blocks

Allow new static scopes to be defined in the midst of executable code.

Allows a section of code to have its own local variables, whose scope is minimized.

Their storage is allocated when the section is entered, and deallocated when the section is exited.

Such section of code is calle a **Block.**

Blocks

- Ex: C based languages allow any compound statement to have declarations and thereby define a new scope

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

Blocks

Scopes that are created by blocks, which could be nested in larger blocks are treated exactly like those created in subprograms

References to variables not declared in the block, are connected to the declarations by searching enclosing scopes, similar to the nested scopes created by nested subprograms.

Dynamic Scope

Dynamic scoping is based on the calling sequence of subprograms, not on their spatial relationship to each other.

Therefore scoping can only be determined at run-time.

Dynamic Scope

Scope can be determined only at run time.

Dynamic scoping requires keeping knowledge of names at runtime.

Names refer to values in the most recently entered scope.

The behaviour of function can be changed without recompiling.

Example:

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

Dynamic Scope

In dynamic scoping how the correct meaning be determined?

- Search with the local declarations.
- If this fails, the variable declaration of the immediate **dynamic parent (calling function)** is searched.
- Perform the previous step for all dynamic ancestors till the declaration found.
- If the declaration is not found, generates a run-time error.

Scope and Lifetime

Sometimes scope and lifetime appear to be related

Ex: A method local variable in a Java method with no method calls inside it

- Lifetime starts and ends with the execution of the method
- Scope is from the declaration statement to the end of method

But, the scope is a spatial concept, lifetime is a temporal concept.

Referencing Environment

The referencing environment of a statement is the 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.

In such a language, the referencing environment of a statement is needed while that statement is being compiled, so code and data structures can be created to allow references to variables from other scopes during run time.

Named Constants

A variable that is bound to a value only once.

Named constants enhance readability and the reliability of a program.

They can be used to parameterize a program.

C++ allows dynamic binding of values to named constants.

```
const int result=2 * width + 1;
```

Summary