University of Southampton

Variables

COMP2322 Programming Language Concepts

Dr Nicholas Gibbins
(based on slides by Dr Julian Rathke)

Variables

The original meaning of variable is that of a reference to a memory location whose contents may change

- Now generalised as "a placeholder for a value of some possibly complex type"
- e.g. in functional languages variables can store closures of arbitrary higher-order types like `((int → int) → int)`
- This is a semantic concept - not only a memory location

Where variables do refer to explicit memory locations, some languages allow you to obtain the location information

- `&var` in C takes variables to pointers, the address of the variable in virtual memory.
- In Java, you can use the `Unsafe` class (but really shouldn't)

The term **aliasing** refers to two variables pointing to the same memory location.

- This situation is generally wished to be avoided - why?

Variables

A variable typically has six attributes associated with it:

- A **name**
- An **address** (aka an L-Value, i.e. the left hand side of an assignment)
- A **value** (aka an R-Value, i.e. the right hand side of an assignment)
- A **type**
- An **extent**
- A **scope**

Names

Names (also referred to as identifiers) are essential in programming languages - we use them to identify the virtual entities that we manipulate in programs

There are ad hoc design choices used in what is acceptable as a name

- Some languages are case sensitive, others not
- Some languages have restricted or fixed length names
- Some enforce lexical rules, e.g. must begin with an alpha, or must contain only alphanumerics, or may not contain certain characters
- Sometimes names can clash with reserved words (keywords) in languages, some languages forbid this

It is not presently clear where a canonical choice for naming schemes would come from, so we are perhaps stuck with these ad hoc choices for a while

Binding

A **binding** is an association between an entity and some attribute

- e.g. between a variable and its type
- or between a variable and its scope

Does a runtime system need to know about the type of a variable?

- What may happen at runtime that necessitates this?
- Some entities do not need to be bound at runtime
- Some entities must be bound at runtime

It is useful to make a distinction between **static** and **dynamic** binding

Static vs. dynamic binding

Static binding occurs before execution (compile time) and remains unchanged throughout execution

Dynamic binding first occurs during execution (runtime) or changes during execution (runtime)

Allocation and Deallocation

One of a variable's attributes is its **address**

- We refer to the binding of a variable to its address as **allocation**
- In complement, we refer to the unbinding of a variable to its address as **deallocation**

Allocation can be static (initialisation time) or dynamic (during runtime)

Deallocation is largely a dynamic concept

A variable's **extent** is the time between its allocation and deallocation

Four kinds of variables

Static variables (aka global variables)

- Bound to a memory location at initialisation time
- e.g. Static class variables in Java are static variables

Stack-dynamic variables (aka local variables)

- Memory is allocated from a runtime stack and bound when a declaration is executed and deallocated when the procedure block it is contained in returns.
- e.g. Local variables in a method declaration

Four kinds of variables

Explicit heap-dynamic variables

- Nameless abstract memory locations that are allocated/deallocated by explicit runtime commands by the programmer.
- e.g. malloc/free in C, new/delete in C++, all objects in Java using new()

Implicit heap-dynamic variables

- Memory in heap is allocated when values are assigned to variables. It is deallocated and reallocated with re-assignment. Error prone and inefficient.
- Used in ALGOL 68, LISP, C and JavaScript (for arrays)

Static type binding

Two approaches to static type binding:

Type declaration

- Most commonly used approach (used in Algol, Pascal, Ada, Cobol, C/C++, Java)
- A variable is introduced with an explicit type and possibly an initial value

Type inference

- No types in variable declarations; the type is inferred from the usage of the variable or by following a fixed naming scheme
- Primitive type inference (arguably another form of explicit declaration) - e.g. in Fortran I, J, K, L, M and N are Integer types, otherwise Real assumed. In Perl $p is a number or a string, @p an array, %p a hash
- More sophisticated - Hindley-Milner inference introduced in ML has few annotations and the compiler deduces a most general type for a variable by its usage. The most general type is typically expressed using polymorphism or generics.

Dynamic type binding

Dynamic type binding typically occurs as a variable is assigned to a value at runtime

- A variable's type binding can change during execution simply by assigning to it a value of a different type
- Commonly used in scripting languages such as JavaScript, Lua, Perl, PHP, Python, Ruby
- Efficiency implications (both time and space) due to runtime type checking
- Arguably advantages in readability and coding convenience

Extent

The **extent** (or lifetime) of a variable refers to the periods of execution time for which it is bound to a particular location storing a meaningful value

Extent is a semantic concept and depends on the execution model

A running program may enter and leave a given extent many times, as in the case of a closure.

Extent

Different kinds of variables have different extents:

- **Static variables** have an extent of whole program execution
- **Stack-dynamic variables** have an extent of a particular stack frame or procedure call
- **Explicit heap-dynamic variables** have an extent from explicit allocation to explicit deallocation (cf. garbage collection and memory leaks)
- **Implicit heap-dynamic variables** have an extent from implicit allocation to implicit deallocation (values may persist in memory but addresses are freed)

Scope

The **scope** of a variable is the part of the code in which it can be referenced

Alternatively, it is the part of a program where a variable's name is meaningful

A variable's scope affects its extent. A no-longer referenceable value may be considered as a meaningless value. Garbage collectors are based on this principle.

Scope

Local variables are declared within a program block; the block is the scope of the variable

Static variables have whole program scope, except where they are temporarily hidden by a locally scoped variable with the same name

We refer to **lexical scope** where scope is aligned to statically determined areas of source code e.g. a class definition, a code block, or method body

- A lexical concept, not a semantic concept

Dynamic scope*

In contrast to lexical scope, some languages support **dynamic scope** for variables.

Dynamic scope is determined at runtime only as it depends on control flow.

Imagine a stack of value bindings for each variable that is updated with the control stack.

A variable is in a dynamic scope if its name is meaningful within the bindings of the current call stack.

* not really scope

Dynamic scope

Dynamic scope is uncommon in modern programming languages as it flies in the face of referential transparency.

It is however used in Perl and Lisp

Example:

- y is lexically scoped and is local to `first`
- x is dynamically scoped and is still in scope when calling `second()`
- If `second()` were called not via `first()` then x would not be in scope

```
first();

sub first {
    local $x = 1;
    my $y=1;
    second();
}

sub second {
    print "x=", $x, "\n";
    print "y=", $y, "\n";
}
```

Scope in ECMAScript

```
// global scope
var a = 1;

function one() {
    alert(a);
}

one()
```

```
outputs '1'
```

Scope in ECMAScript

```
// local scope
function two(a) {
    alert(a);
}

two(2)
```

```
outputs '2'
```

Scope in ECMAScript

```
// local scope again
function three() {
    var a = 3;
    alert(a);
}

three()
```

```
outputs '3'
```

Scope in ECMAScript

```
// no block scope in ECMAScript
function four() {
    if (true) {
        var a = 4;
    }
    alert(a);
}

four()
```

```
outputs '4'
```

http://stackoverflow.com/questions/500431/javascript-variable-scope

Scope in ECMAScript

```
function Five() {
    this.a = 5;
}

alert(new Five().a);
```

```
outputs '5'
```

Scope in ECMAScript

```
var six = (function() {
    var a = 6;
    return function() {
        // ECMAScript "closure" means I have access to 'a' in here,
        // because it is defined in the function in which I was
        // defined.
        alert(a);
    };
})();

six()
```

```
outputs '6'
```

http://stackoverflow.com/questions/500431/javascript-variable-scope

Scope in ECMAScript

```
function Seven() {
    this.a = 7;
}

// [object].prototype.property loses to
// [object].property in the lookup chain. For example...

// Not reached, because 'a' is set in the constructor above.
Seven.prototype.a = -1;

// Reached, even though 'b' is NOT set in the constructor.
Seven.prototype.b = 8;

alert(new Seven().a);
alert(new Seven().b);
```

```
outputs '7'
outputs '8'
```

http://stackoverflow.com/questions/500431/javascript-variable-scope

Scope in OCaml

Parameters p1, p2 in scope in whole function body

```
let add_polynom p1 p2 =
  let n1 = Array.length p1
  and n2 = Array.length p2 in
    let result = Array.create(max n1 n2) 0 in
    for i = 0 to n1 - 1 do result.(i) <- p1.(i) done;
    for i = 0 to n2 - 1 do result.(i) <- result.(i) + p2.(i) done;
    result;;
```

Local variables n1, n2 in scope after ‘in’

Scope in OCaml

Recursive local variable fact in scope even before ‘in’

```
let rec fact n = n * fact (n - 1 ) in
   fact (10);;
```

University of Southampton

Next Lecture:
Syntax and Grammars