SIENA*college*
Computer Science

Computer Science 340
Programming Languages
Siena College
Fall 2019

Topic Notes: Names and Bindings

We now turn our attention to the naming and binding of variables to memory locations in programming languages. We will look more carefully at some familiar concepts such as what are legal names, and what are the types, lifetimes, and scopes of variables.

Variables

We use variables in our programs all the time, usually without thinking much about it.

A variable is an abstraction of one or more memory locations. It has attributes including

- a name
- an address (its "L-value")
- a value (its "R-value")
- a datatype
- a lifetime
- a scope

Names

Names for variables (and other identifiers in our programs) are subject to some important design decisions:

- what characters can be used in names?
- are names case sensitive?
- are some names unavailable for general use (reserved words/keywords)?

If names are restricted to be too short, they cannot be as meaningful.

Some examples:

- Fortran 95: maximum of 31

- C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31 (though gcc does not seem to suffer from this limit)
- C#, Ada, and Java: no limit, and all are significant
- C++: no limit, but implementers often impose one
- Some simple BASIC implementations limit to 1 or 2 characters (!)

Some languages also place further meaning on special characters within a name:

- PHP: all variable names must begin with dollar signs
- Perl: all variable names begin with special characters, which specify the variable's type
- Ruby: variable names that begin with `@` are instance variables; those that begin with `@@` are class variables, variable names that begin with `$` are globals
- BASIC: variable names that end in `$` are strings

See Example:
`/home/cs340/examples/names/perl_names.pl`

Names in C-based languages are case sensitive, but many other languages are not. This can cause some problems with multi-language programming: *e.g.*, calling a Fortran subroutine from C.

See Example:
`/home/cs340/examples/names/caseinsensitive.f95`

Special names include *keywords* and *reserved words*.

- Keywords are special only in some contexts, and may be used for other purposes in other contexts.
- Reserved words can never be used except for their "special" purpose. Almost all languages treat all of the special words this way.
- A language may wish to limit the number of reserved words (COBOL has 400!).

However, the terms are often used interchangeably.

On the web: C++ Keywords at
`http://en.cppreference.com/w/cpp/keyword`

Addresses

Any variable needs to have some address associated with it where it can store its value.

- A variable may have different addresses at different times during execution
- A variable may have different addresses at different places in a program
- If two variable names can be used to access the same memory location, they are called *aliases*
- Aliases are created via pointers, reference variables, C and C++ unions
- Aliases are harmful to readability (program readers must remember all of them)

In C:

```
int a[10];
int *b = a;
```

Here, `a` and `b` will appear to the reader of the program as two different arrays, but they are really just two names for the same array.

Type

The type, or datatype, of a variable determines the range of values of variables and the set of operations that are defined for values of that type.

For floating point data, the type also determines the precision (*i.e.*, `float` *vs.* `double`).

Binding

Binding is the association of a particular attribute to an entity or an operation to a symbol.

Binding Time: when does this binding occur?

- *static binding*
  - takes place at compile time
  - remains the same throughout the execution of the program
- *dynamic binding*
  - binding first done at runtime
  - binding may change during program execution

Or, looking at a wider time scale:

- language design time – bind operator symbols to operations

- language implementation time – bind floating point type to a specific representation
- compile time – bind a variable to a type in C or Java
- load time – bind a C or C++ `static` variable to a memory cell
- run time – bind a nonstatic local variable to a memory cell

---

Type Bindings

Type bindings – the assignment of a datatype to a variable – is static for most languages.

In some languages, we accomplish the static binding when we declare a variable – we must give it a type in addition to a name.

```
double d;
```

In other languages, the type bindings are implicit. For example, we saw that BASIC variables are numeric unless the name ends in a `$`. In FORTRAN 77, unless `IMPLICIT NONE` is specified, variable identifiers are assumed to be type `REAL` if they start with the letters `A-H` and `O-Z`, and `INTEGER` if they start with the letters `I-N`.

This approach provides a small amount of convenience, but can lead to problems with reliability (*e.g.*, the use of an implicit variable when a different variable was intended).

With dynamic type binding, we do not need to specify a datatype at declaration.

In some cases, a type is bound to the variable when the variable is assigned a value. For example, in JavaScript, we can declare a variable such as:

```
var x;
```

and later give it a value:

```
x = 7;
```

and that would result in the datatype of `x` being an integer.

Later in the same program, we could reassign that variable to something else:

```
x = [17.23, 9.1];
```

and `x` is now a list of floating point values.

See Example:
`/home/cs340/examples/names/js_binding.html`

The necessitates runtime checking of datatypes to ensure operations on those values are appropriate for the actual datatype. It also means we limit the errors that can be detected by the compiler.

Other languages use *type inferencing*. Here, the language infers the datatype of variable based on elements involved in the expression.

For a *strongly typed language*, inference makes many type declarations unnecessary.

Type checking is the activity of ensuring that the operands of an operator are of compatible types.

- *type coercion* occurs when an operand is converted to a type applicable to the current operation
- *type errors* occur if an operator is applied to an operand of an invalid type
- a language is called strongly typed if all type errors can be caught at compile time
- type coercion weakens the typing system of a language

Storage Bindings

The assignment of memory locations to variables is called *storage binding*. Key ideas here:

- *Allocation* – the binding of a memory cell to a variable
- *Deallocation* – returning a cell back into the available memory pool
- *Lifetime* of a variable – the time during which the variable is bound to a specific memory location

We can categorize variables by their lifetimes:

- *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

  See Example:
  `/home/cs340/examples/names/static.c`

  - advantages: efficiency (direct addressing), history-sensitive subprogram support
  - disadvantage: lack of flexibility (no recursion)
  - similar: *class variables* in C++, Java.

- *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.)

  - *local variables* in C subprograms (not declared `static`) and Java methods

  - allocated on the *run-time stack*
  - advantage: allows recursion; conserves storage
  - disadvantages: overhead of allocation and deallocation, subprograms cannot be history sensitive, inefficient references (indirect addressing)

- *Explicit heap-dynamic* – allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
  - referenced only through pointers or references, *e.g.*, dynamic objects in C++ (via `new` and `delete`), all objects in Java
  - allocated from the *heap*
  - advantage: provides for dynamic storage management
  - disadvantage: inefficient and unreliable

- *Implicit heap-dynamic* – allocation and deallocation caused by assignment statements
  - all variables in APL; all strings and arrays in Perl, JavaScript, and PHP
  - advantage: flexibility (generic code)
  - disadvantages: inefficient, because all attributes are dynamic, loss of error detection

---

Scope

The *scope* of a variable is the range of statements over which it is *visible*. That is, the range of statements where the variable's bindings apply (*i.e.*, where we can use it!).

The *local variables* of a program unit are those that are declared in that unit.

The *nonlocal variables* of a program unit are those that are visible in the unit but not declared there. *Global variables* are a special category of nonlocal variables.

The *scope rules* of a language determine how references to names are associated with variables.

There are two major categories: *static* or *lexical scope*, and *dynamic scope*.

With static scope, the scope of a variable depends only on the program text – it can be determined completely at compile time. That is, any name can be connected to a variable declaration (or determined to be undeclared) by the compiler.

See Example:
`/home/cs340/examples/names/JavaScope.java`

We can tell exactly which variables are visible at each line of this Java program. If a name does not match something in the local scope (local variables and parameters), we look at the instance and class variables.

We can summarize the basic rule: a variable is visible until the `}` that matches the `{` most recently preceding the declaration. An important exception occurs in C++, Java, and C#, where variables declared in `for` statements have their scope restricted to the `for` construct.

An exception here is that the instance variables are only accessible in non-`static` methods.

Some names can be *masked* by definitions in enclosed scopes (or *blocks*). C and C++ will allow this at any level (a variable declared local to a `while` loop can mask a local variable within a method/function), while Java does not permit this.

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

The C-based languages (*e.g.*, C99, C++, Java, and C#) allow variable declarations to appear anywhere a statement can appear (in the middle of a block). Older versions of C required that all declarations appeared before any other statements.

In C99, C++, and Java, the scope of all local variables is from the declaration to the end of the block.

In C#, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block. However, a variable still must be declared before it can be used.

Hidden declarations can sometimes be accessed anyway, such as by prepending `this.` before the name of a Java instance variable that has been masked by a local or parameter.

Related to this is the `let` construct in functional languages including Scheme.

See Example:
`/home/cs340/examples/names/let.scm`

The names `x1`, `x2`, `y1`, `y2` act similarly to local variables with a scope that lasts through the remainder of the `let` function. The main difference is that their values cannot change once set.

A better example improves on the `index` function.

See Example:
`/home/cs340/examples/names/betterindex.scm`

Here, we avoid the potential of 2 redundant recursive calls.

This can be even more complex in languages that allow *nested subprogram definitions*, which create nested static scopes. This is the case in several languages including Ada, JavaScript, Common LISP, Scheme, Fortran 2003+, F#, and Python.

See Example:
`/home/cs340/examples/names/js.html`

Here, the functions `sub1` and `sub2` only can be called from elsewhere in the enclosing function `setMessage`. Even though `setMessage` calls `sub1`, which in turn calls `sub2`, the variable `x` in `sub2` refers to the `x` declared in `setMessage`, even though its most recent *dynamic* ancestor is `sub1`.

If the language used *dynamic scoping* instead, the search for nonlocal bindings would follow the call history rather than the static hierarchy.

Global Scope

C, C++, PHP, and Python programs can consist of a sequence of function definitions in a file. These languages allow variable declarations to appear outside function definitions and these are *global variables* accessible to all functions.

A C and C++ global declaration defines both the type and allocates the storage:

```
int sum;
```

To refer to a global variable defined in a different file without allocating space for it:

```
extern int sum;
```

These languages also support a file scope global-like variable that we can access only within the file where it is defined:

```
static int sum;
```

This allows the variable to be shared among the file's functions without the potential for name collisions with other files.

In C++, if a global variable is masked by a local of the same name, it can be preceded by `::` to force access to the global.

In Python, a global variable can be referenced in functions, but can be assigned in a function only if it has been declared to be global in the function. This is necessary because Python automatically implicitly declares local variables when they are assigned. If a reference is intended to be to a global rather than the creation of a new local, it must first be declared to be `global` in the function.

Named Constants

A *named constant* is a variable that is bound to a value only when it is bound to storage. As you know from your programming experience so far, their use enhances program readability and modifiability and can parameterize programs.

The binding of values to named constants can be either static (called *manifest constants*) or dynamic.

In Ada, C++, and Java, the expressions that initialize a constant can be of any kind, dynamically bound (*i.e.*, you can use variables).

C# has two kinds, `readonly` and `const`. The the values of `const` named constants are bound at compile time, while the values of `readonly` named constants are dynamically bound.

C does not support named constants, but `#define` preprocessor directives are often used for the same purpose.