

id: structures title: Language Structures allowDiscussion: None subject: description: Puppet Language Structure Reference contributors: creators: jesse effectiveDate: 2005/08/27 15:06:55.228 GMT-5 expirationDate: None language: en rights: creation_date: 2005/07/29 12:14:07.765 GMT-5 modification_date: 2005/08/27 15:06:55.230 GMT-5 layout: document_view Content-Type: text/x-rst

Language Structures

This is a brief overview of the language structures available for making site configurations in Puppet. For further documentation, visit the Puppet homepage <http://reductivelabs.com/projects/puppet/>

Types

The basic unit of configuration in Puppet are **types**. Types model objects on the computer being managed, and each builtin type has attributes that determine the final type configuration:

```
file { "/etc/passwd": owner => root, mode => 644 }
package { apache: install => true }
```

Puppet also provides facilities for defining new types as collections of existing types (see [Composition](#) below), but there is no syntactic difference between using builtin types like `file` and `package` and using defined types. Any operation or syntax that succeeds for builtin types should also work for defined types.

See the [Type Reference](#) for the documentation for the Puppet Library's primitive types.

Assignment

`$variable = value`

Variables available in the current scope are referenced by preceding them with the `$` character.

Once assigned, variables cannot be reassigned. However, within a sub-scope a new assignment can be made for a variable name for that sub-scope and any further sub-scopes created within it:

```
$x = "foo"
$y = "bar"
$z = $x$y
```

Scope

Generally speaking, any language structure that involves curly braces creates a new scope inside those braces. This currently includes server and class definitions and `if/then/else` structures. Each file should also introduce its own scope but currently does not.

Once assigned, variables cannot be reassigned within the same scope. However, within a sub-scope a new assignment can be made for a variable name for that sub-scope and any further scopes created within it:

```
$var = value

# override $var
define testing {
    $var = othervalue
}
```

Service and class definitions are scoped just as variable assignments are. Functions defined and Classes created within a scope will not be available outside the scope in which they are created:

```

define testing {
  file { "/etc/passwd": owner => root }
}

class osx {
  # override the existing testing definition
  define testing {
    file { "/etc/other": owner => root }
  }
}

```

The evaluation by Puppet of following example would be result in the copying of `/file_repository/test-httpd.conf` to `/etc/httpd/conf/httpd.conf`:

```

$filename = "/etc/apache/httpd.conf"

class webserver {
  $filename = "/etc/httpd/conf/httpd.conf"
  define httpd_service (config_file) {
    file { $filename : source => $config_file }
  }
  httpd_service { "test_httpd" :
    config_file => "/file_repository/test-httpd.conf"
  }
}

webserver {}

```

Composition

```

define <name>(<param1>,<param2>,...) {...}

```

Definition of functions allows the composition of lower level types into higher level types.

Parameters of defined functions can be referenced within the definition scope, similarly to variables, by preceding their names with the `$` character:

```

define ntpd_service (config_source) {
  file { "/etc/ntp.conf": source => $config_source }
  service { ntpd: running => true }
}

ntp_service { "test_ntpd":
  config_source => "/nfs/server/configs/ntp.conf"
}

```

Server Classes

```

class <class_name> [inherits <super_class_name>] { ... }

```

Class definitions allow the specification of a hierarchy of server classes; a host that is a member of a subclass will apply the configuration from the subclass and all parent classes:

```

# really simple example
class solaris {
  file {

```

```

        "/etc/passwd": owner => root, group => root, mode => 644;
        "/etc/shadow": owner => root, group => root, mode => 440
    }
}

class solworkstation inherits solaris {
    file {
        "/etc/sudoers": owner => root, group => root, mode => 440;
        "/bin/sudo": owner => root, group => root, mode => 4111
    }
}

```

Conditionals

Puppet currently supports two types of conditionals: in-statement and around statements. We call the in-statement conditionals **selectors**, as they are essentially a select-style operator, which support the use of **default** to specify a default value:

```

define testing(os) {
    owner = $os ? {
        sunos => adm,
        redhat => bin,
        default => root
    }
    file { "/some/file": owner => $owner }
}

```

case provides the ability to conditionally apply types:

```

case $operatingsystem {
    sunos:    { solaris {} # apply the solaris class }
    redhat:   { redhat {} # apply the redhat class }
    default:  { generic {} # apply the generic class }
}

```

Bringing Config files together

import <filename>

Starts the parsing of the file specified and creates any specified definitions and classes at the current scope. Currently files are only searched for within the same directory as the file doing the importing. We expect to add a search path soon.

Reserved words

Generally, any word that the syntax uses for special meaning is probably also a reserved word, meaning you cannot use it for variable or type names. Thus, words like **true**, **define**, **inherits**, and **class** are all reserved.

Comments

Puppet supports sh-style comments; they can either be on their own line or at the end of a line (see the [Conditionals](#) example above).