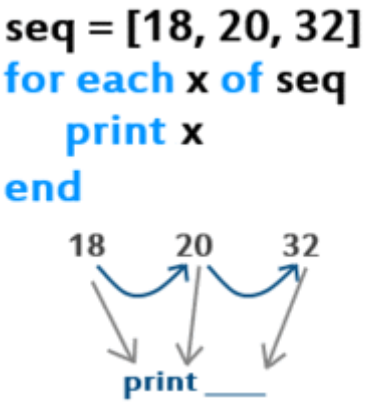
Foreach loop

Foreach loop (or **for each loop**) is a control flow statement for traversing items in a collection. Foreach is usually used in place of a standard **for loop** statement. Unlike other for loop constructs, however, foreach loops[1] usually maintain no explicit counter: they essentially say "do this to everything in this set", rather than "do this *x* times". This avoids potential off-by-one errors and makes code simpler to read. In object-oriented languages an iterator, even if implicit, is often used as the means of traversal.

The *foreach* statement in some languages has some defined order, processing each item in the collection from the first to the last. The *foreach* statement in many other languages, especially array programming languages, does not have any particular order. This simplifies loop optimization in general and in particular allows vector processing of items in the collection concurrently.



For each loops are almost always used to iterate over items in a sequence of elements.

Contents

Syntax

Syntax varies among languages. Most use the simple word `for`, roughly as follows:

```
for each item in collection:
  do something to item
```

Language support

Programming languages which support foreach loops include ABC, ActionScript, Ada, C++11, C#, ColdFusion Markup Language (CFML), Cobra, D, Daplex (query language), Delphi, ECMAScript, Erlang, Java (since 1.5), JavaScript, Lua, Objective-C (since 2.0), ParaSail, Perl, PHP, Prolog,[2] Python, REALbasic, Ruby, Scala, Smalltalk, Swift, Tcl, tcsh, Unix shells, Visual Basic .NET, and Windows PowerShell. Notable languages without foreach are C, and C++ pre-C++11.

ActionScript 3.0

ActionScript supports the ECMAScript 4.0 Standard[3] for `for each .. in`[4] which pulls the value at each index.

```
var foo:Object = {
    "apple":1,
    "orange":2
};

for each (var value:int in foo) {
    trace(value);
}

// returns "1" then "2"
```

It also supports `for .. in`[5] which pulls the key at each index.

```
for (var key:String in foo) {
    trace(key);
}

// returns "apple" then "orange"
```

Ada

Ada supports foreach loops as part of the normal for loop. Say X is an array:

```
for I in X'Range loop
   X (I) := Get_Next_Element;
end loop;
```

This syntax is used on mostly arrays, but will also work with other types when a full iteration is needed.

Ada 2012 has generalized loops to foreach loops on any kind of container (array, lists, maps...):

```
for Obj of X loop
   -- Work on Obj
end loop;
```

C

The C language does not have collections or a foreach construct. However, it has several standard data structures that can be used as collections, and foreach can be made easily with a macro.

However, two obvious problems occur:

- The macro is unhygienic: it declares a new variable in the existing scope which remains after the loop.
- One foreach macro cannot be defined that works with different collection types (e.g., array and linked list) or that is extensible to user types.

C string as a collection of char

```c
#include <stdio.h>

/* foreach macro viewing a string as a collection of char values */
#define foreach(ptrvar, strvar) \
char* ptrvar; \
for (ptrvar = strvar; (*ptrvar) != '\0'; *ptrvar++)

int main(int argc, char** argv) {
 char* s1 = "abcdefg";
 char* s2 = "123456789";
 foreach (p1, s1) {
  printf("loop 1: %c\n", *p1);
 }
 foreach (p2, s2) {
  printf("loop 2: %c\n", *p2);
 }
 return 0;
}
```

C int array as a collection of int (array size known at compile-time)

```c
#include <stdio.h>

/* foreach macro viewing an array of int values as a collection of int values */
#define foreach(intpvar, intarr) \
int* intpvar; \
for (intpvar = intarr; intpvar < (intarr + (sizeof(intarr)/sizeof(intarr[0]))); ++intpvar)

int main(int argc, char** argv) {
 int a1[] = {1, 1, 2, 3, 5, 8};
 int a2[] = {3, 1, 4, 1, 5, 9};
 foreach (p1, a1) {
  printf("loop 1: %d\n", *p1);
 }
 foreach (p2, a2) {
  printf("loop 2: %d\n", *p2);
 }
 return 0;
}
```

Most general: string or array as collection (collection size known at run-time)

Note: `idxtype` can be removed and `typeof(col[0])` used in its place with GCC

```c
#include <stdio.h>
#include <string.h>

/* foreach macro viewing an array of given type as a collection of values of given type */
#define arraylen(arr) (sizeof(arr)/sizeof(arr[0]))
#define foreach(idxtype, idxpvar, col, colsiz) \
idxtype* idxpvar; \
for (idxpvar = col; idxpvar < (col + colsiz); ++idxpvar)

int main(int argc, char** argv) {
 char* c1 = "collection";
 int c2[] = {3, 1, 4, 1, 5, 9};
 double* c3;
 int c3len = 4;
 c3 = (double*)calloc(c3len, sizeof(double));
 c3[0] = 1.2; c3[1] = 3.4; c3[2] = 5.6; c3[3] = 7.8;

 foreach (char, p1, c1, strlen(c1)) {
  printf("loop 1: %c\n", *p1);
 }
 foreach (int, p2, c2, arraylen(c2)) {
  printf("loop 2: %d\n", *p2);
 }
 foreach (double, p3, c3, c3len) {
  printf("loop 3: %.1lf\n", *p3);
```

```
 }
 return 0;
}
```

C#

In C#, assuming that myArray is an array of integers:

```
foreach (int x in myArray) { Console.WriteLine(x); }
```

Language Integrated Query (LINQ) provides the following syntax, accepting a delegate or lambda expression:

```
myArray.ToList().ForEach(x => Console.WriteLine(x));
```

C++

C++11 provides a foreach loop. The syntax is similar to that of Java:

```
#include <iostream>

int main()
{
  int myint[] = {1, 2, 3, 4, 5};

  for (int i : myint)
  {
    std::cout << i << '\n';
  }
}
```

C++11 range-based for statements have been implemented in GNU Compiler Collection (GCC) (since version 4.6), Clang (since version 3.0) and Visual C++ 2012 (version 11 [6])

The range-based `for` is syntactic sugar equivalent to:

```
  for (auto __anon = begin(myint); __anon != end(myint); ++__anon)
  {
    auto i = *__anon;
    std::cout << i << '\n';
  }
```

The compiler uses argument-dependent lookup to resolve the `begin` and `end` functions.[7]

The C++ Standard Library also supports `for_each`,[8] that applies each element to a function, which can be any predefined function or a lambda expression. While range-based for is only from the beginning to the end, the range and direction you can change the direction or range by altering the first two parameters.

```
#include <iostream>
#include <algorithm> // contains std::for_each
#include <vector>

int main()
{
  std::vector<int> v {1, 2, 3, 4, 5};
```

```
  std::for_each(v.begin(), v.end(), [&](int i)
  {
    std::cout << i << '\n';
  });

  std::cout << "reversed but skip 2 elements:\n";

  std::for_each(v.rbegin()+2, v.rend(), [&](int i)
  {
    std::cout << i << '\n';
  });
}
```

Qt, a C++ framework, offers a macro providing foreach loops[9] using the STL iterator interface:

```
#include <QList>
#include <QDebug>

int main()
{
  QList<int> list;

  list << 1 << 2 << 3 << 4 << 5;

  foreach (int i, list)
  {
    qDebug() << i;
  }
}
```

Boost, a set of free peer-reviewed portable C++ libraries also provides foreach loops:[10]

```
#include <boost/foreach.hpp>
#include <iostream>

int main()
{
  int myint[] = {1, 2, 3, 4, 5};

  BOOST_FOREACH(int &i, myint)
  {
    std::cout << i << '\n';
  }
}
```

C++/CLI

The C++/CLI language proposes a construct similar to C#.

Assuming that myArray is an array of integers:

```
    for each (int x in myArray)
    {
        Console::WriteLine(x);
    }
```

ColdFusion Markup Language (CFML)

Script syntax

```
// arrays
arrayeach([1,2,3,4,5], function(v){
    writeOutput(v);
});

// or

for (v in [1,2,3,4,5]){
    writeOutput(v);
}

// or

// (Railo only; not supported in ColdFusion)
letters = ["a","b","c","d","e"];
letters.each(function(v){
    writeOutput(v); // abcde
});

// structs
for (k in collection){
    writeOutput(collection[k]);
}

// or

structEach(collection, function(k,v){
    writeOutput("key: #k#, value: #v#;");
});

// or
// (Railo only; not supported in ColdFusion)
collection.each(function(k,v){
    writeOutput("key: #k#, value: #v#;");
});
```

Tag syntax

```
<!--- arrays --->
<cfloop index="v" array="#['a','b','c','d','e']#">
  <cfoutput>#v#</cfoutput><!--- a b c d e  --->
</cfloop>
```

CFML incorrectly identifies the value as "index" in this construct; the `index` variable does receive the actual value of the array element, not its index.

```
<!--- structs --->
<cfloop item="k" collection="#collection#">
    <cfoutput>#collection[k]#</cfoutput>
</cfloop>
```

Common Lisp

Common Lisp provides foreach ability either with the *dolist* macro:

```
(dolist (i '(1 3 5 6 8 10 14 17))
  (print i))
```

or the powerful *loop* macro to iterate on more data types

```
(loop for i in '(1 3 5 6 8 10 14 17)
      do (print i))
```

and even with the *mapcar* function:

```
(mapcar #'print '(1 3 5 6 8 10 14 17))
```

D

```
foreach(item; set) {
  // do something to item
}
or
foreach(argument) {
  // pass value
}
```

Dart

```
for (final element in someCollection) {
  // do something with element
}
```

Object Pascal, Delphi

Foreach support was added in Delphi 2005, and uses an enumerator variable that must be declared in the *var* section.

```
for enumerator in collection do
begin
  //do something here
end;
```

Eiffel

The iteration (foreach) form of the Eiffel loop construct is introduced by the keyword `across`.

In this example, every element of the structure `my_list` is printed:

```
            across my_list as ic loop print (ic.item) end
```

The local entity `ic` is an instance of the library class `ITERATION_CURSOR`. The cursor's feature `item` provides access to each structure element. Descendants of class `ITERATION_CURSOR` can be created to handle specialized iteration algorithms. The types of objects that can be iterated across (`my_list` in the example) are based on classes that inherit from the library class `ITERABLE`.

The iteration form of the Eiffel loop can also be used as a boolean expression when the keyword `loop` is replaced by either `all` (effecting universal quantification) or `some` (effecting existential quantification).

This iteration is a boolean expression which is true if all items in `my_list` have counts greater than three:

```
            across my_list as ic all ic.item.count > 3 end
```

The following is true if at least one item has a count greater than three:

```
            across my_list as ic some ic.item.count > 3 end
```

Go

Go's foreach loop can be used to loop over an array, slice, string, map, or channel.

Using the two-value form, we get the index/key (first element) and the value (second element):

```
for index, value := range someCollection {
    // Do something to index and value
}
```

Using the one-value form, we get the index/key (first element):

```
for index := range someCollection {
    // Do something to index
}
```

[11]

Groovy

Groovy supports *for* loops over collections like arrays, lists and ranges:

```
def x = [1,2,3,4]
for (v in x)            // loop over the 4-element array x
{
    println v
}

for (v in [1,2,3,4])    // loop over 4-element literal list
{
    println v
}

for (v in 1..4)         // loop over the range 1..4
{
    println v
}
```

Groovy also supports a C-style for loop with an array index:

```
for (i = 0; i < x.size(); i++)
{
    println x[i]
}
```

Collections in Groovy can also be iterated over using the *each* keyword and a closure. By default, the loop dummy is named *it*

```
x.each{ println it }     // print every element of the x array
x.each{i-> println i}    // equivalent to line above, only loop dummy explicitly named "i"
```

Haskell

Haskell allows looping over lists with monadic actions using `mapM_` and `forM_` (`mapM_` with its arguments flipped) from Control.Monad (http://hackage.haskell.org/package/base-4.6.0.1/docs/Control-Monad.html):

code	prints

`mapM_ print [1..4]`	`1`
	`2`
	`3`
	`4`

`forM_ "test" $ \char -> do`
`  putChar char`
`  putChar char`
	`tteesstt`

It's also possible to generalize those functions to work on applicative functors rather than monads and any data structure that is traversable using `traverse` (`for` with its arguments flipped) and `mapM` (`forM` with its arguments flipped) from Data.Traversable (http://hackage.haskell.org/package/base-4.6.0.1/docs/Data-Traversable.html).

Haxe

```
for (value in iterable) {
    trace(value);
}

Lambda.iter(iterable, function(value) trace(value));
```

Java

In Java, a foreach-construct was introduced in Java Development Kit (JDK) 1.5.0.[12]

Official sources use several names for the construct. It is referred to as the "Enhanced for Loop",[12] the "For-Each Loop",[13] and the "foreach statement".[14]

```
for (Type item : iterableCollection) {
    // Do something to item
}
```

JavaScript

The EcmaScript 6 standard has `for..of (https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/for...of)` for index-less iteration over generators, arrays and more:

```
for (var item of array){
    // Do stuff
}
```

Alternatively, function-based style: [15]

```
array.forEach(item => {
    // Do stuff
})
```

For unordered iteration over the keys in an Object, JavaScript features the `for...in` loop:

```
for (var key in object) {
    // Do stuff with object[key]
}
```

To limit the iteration to the object's own properties, excluding those inherited through the prototype chain, it is sometimes useful to add a hasOwnProperty() test, if supported by the JavaScript engine (for WebKit/Safari, this means "in version 3 or later").

```
for (var key in object) {
    if (object.hasOwnProperty(key)) {
        // Do stuff with object[key]
    }
}
```

ECMAScript 5 provided Object.keys method, to transfer the own keys of an object into array.[16]

```
var book = { name: "A Christmas Carol", author: "Charles Dickens" };
for(var key of Object.keys(book)){
    alert("PropertyName = " key + " Property Value = " + book[key]);
}
```

Lua[17]

Iterate only through numerical index values:

```
for index, value in ipairs(array) do
    -- do something
end
```

Iterate through all index values:

```
for index, value in pairs(array) do
    -- do something
end
```

Mathematica

In Mathematica, `Do` will simply evaluate an expression for each element of a list, without returning any value.

```
In[]:= Do[doSomethingWithItem, {item, list}]
```

It is more common to use `Table`, which returns the result of each evaluation in a new list.

```
In[]:= list = {3, 4, 5};

In[]:= Table[item^2, {item, list}]
Out[]= {9, 16, 25}
```

MATLAB

```
for item = array
%do something
end
```

Mint

For each loops are supported in Mint, possessing the following syntax:

```
for each element of list
    /* 'Do something.' */
end
```

The `for (;;)` or `while (true)` infinite loop in Mint can be written using a for each loop and an infinitely long list.[18]

```
import type
/* 'This function is mapped to'
 * 'each index number i of the'
 * 'infinitely long list.'
 */
sub identity(x)
    return x
end
/* 'The following creates the list'
 * '[0, 1, 2, 3, 4, 5, ..., infinity]'
 */
infiniteList = list(identity)
for each element of infiniteList
    /* 'Do something forever.' */
end
```

Objective-C

Foreach loops, called Fast enumeration, are supported starting in Objective-C 2.0. They can be used to iterate over any object that implements the NSFastEnumeration protocol, including NSArray, NSDictionary (iterates over keys), NSSet, etc.

```
NSArray *a = [NSArray new];       // Any container class can be substituted

for(id obj in a) {                // Note the dynamic typing (we do not need to know the
                                  // Type of object stored in 'a'.  In fact, there can be
                                  // many different types of object in the array.

    printf("%s\n", [[obj description] UTF8String]);  // Must use UTF8String with %s
    NSLog(@"%@", obj);                               // Leave as an object
}
```

NSArrays can also broadcast a message to their members:

```
NSArray *a = [NSArray new];

[a makeObjectsPerformSelector:@selector(printDescription)];
```

Where blocks are available, an NSArray can automatically perform a block on every contained item:

```
[myArray enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop)
    {
        NSLog(@"obj %@", obj);
        if ([obj shouldStopIterationNow])
            *stop = YES;
    }];
```

The type of collection being iterated will dictate the item returned with each iteration. For example:

```
NSDictionary *d = [NSDictionary new];

for(id key in d) {
    NSObject *obj = [d objectForKey:key];      // We use the (unique) key to access the
(possibly nonunique) object.
    NSLog(@"%@", obj);
}
```

OCaml

OCaml is a functional language. Thus, the equivalent of a foreach loop can be achieved as a library function over lists and arrays.

For lists:

```
List.iter (fun x -> print_int x) [1;2;3;4];;
```

or in short way:

```
List.iter print_int [1;2;3;4];;
```

For arrays:

```
Array.iter (fun x -> print_int x) [|1;2;3;4|];;
```

or in short way:

```
Array.iter print_int [|1;2;3;4|];;
```

ParaSail

The ParaSail parallel programming language supports several kinds of iterators, including a general "for each" iterator over a container:

```
var Con : Container<Element_Type> := ...
// ...
for each Elem of Con concurrent loop  // loop may also be "forward" or "reverse" or unordered
(the default)
  // ... do something with Elem
end loop
```

ParaSail also supports filters on iterators, and the ability to refer to both the key and the value of a map. Here is a forward iteration over the elements of "My_Map" selecting only elements where the keys are in "My_Set":

```
var My_Map : Map<Key_Type => Univ_String, Value_Type => Tree<Integer>> := ...
const My_Set : Set<Univ_String> := ["abc", "def", "ghi"];

for each [Str => Tr] of My_Map {Str in My_Set} forward loop
   // ... do something with Str or Tr
end loop
```

Pascal

In Pascal, ISO standard 10206:1990 introduced iteration over set types, thus:

```
var
  elt: ElementType;
  eltset: set of ElementType;

{...}

for elt in eltset do
  { ... do something with elt }
```

Perl

In Perl, *foreach* (which is equivalent to the shorter for) can be used to traverse elements of a list. The expression which denotes the collection to loop over is evaluated in list-context and each item of the resulting list is, in turn, aliased to the loop variable.

List literal example:

```
foreach (1, 2, 3, 4) {
    print $_;
}
```

Array examples:

```
foreach (@arr) {
    print $_;
}
```

```
foreach $x (@arr) { #$x is the element in @arr
    print $x;
}
```

Hash example:

```
foreach $x (keys %hash) {
    print $x . " = " . $hash{$x}; # $x is a key in %hash and $hash{$x} is its value
}
```

Direct modification of collection members:

```
@arr = ( 'remove-foo', 'remove-bar' );
foreach $x (@arr){
    $x =~ s/remove-//;
}
# Now @arr = ('foo', 'bar');
```

PHP

```
foreach ($set as $value) {
    // Do something to $value;
}
```

It is also possible to extract both keys and values using the alternate syntax:

```
foreach ($set as $key => $value) {
    echo "{$key} has a value of {$value}";
}
```

Direct modification of collection members:

```
$arr = array(1, 2, 3);
foreach ($arr as &$value) { // Note the &, $value is a reference to the original value inside
$arr
    $value++;
}
// Now $arr = array(2, 3, 4);

// also works with the full syntax
foreach ($arr as $key => &$value) {
    $value++;
}
```

- More information (https://php.net/foreach)

Python

```
for item in iterable_collection:
    # Do something with item
```

Python's tuple assignment, fully available in its foreach loop, also makes it trivial to iterate on (key, value) pairs in associative arrays:

```
for key, value in some_dict.items():  # Direct iteration on a dict iterates on its keys
    # Do stuff
```

As `for ... in` is the only kind of for loop in Python, the equivalent to the "counter" loop found in other languages is...

```
for i in range(len(seq)):
    # Do something to seq[i]
```

... though using the `enumerate` function is considered more "Pythonic":

```
for i, item in enumerate(seq):
    # Do stuff with item
    # Possibly assign it back to seq[i]
```

Racket

```
(for ([item set])
  (do-something-with item))
```

or using the conventional Scheme `for-each` function:

```
(for-each do-something-with a-list)
```

`do-something-with` is a one-argument function.

Raku

In Raku, a sister language to Perl, *for* must be used to traverse elements of a list (*foreach* is not allowed). The expression which denotes the collection to loop over is evaluated in list-context, but not flattened by default, and each item of the resulting list is, in turn, aliased to the loop variable(s).

List literal example:

```
for 1..4 {
    .say;
}
```

Array examples:

```
for @arr {
    .say;
}
```

The for loop in its statement modifier form:

```
.say for @arr;
```

```
for @arr -> $x {
    say $x;
}
```

```
for @arr -> $x, $y {    # more than one item at a time
    say "$x, $y";
}
```

Hash example:

```
for keys %hash -> $key {
    say "$key: $hash{$key}";
}
```

or

```
for %hash.kv -> $key, $value {
    say "$key: $value";
}
```

or

```
for %hash -> $x {
    say "$x.key(): $x.value()";    # Parentheses needed to inline in double quoted string
}
```

Direct modification of collection members with a doubly pointy block, <->:

```
my @arr = 1,2,3;
for @arr <-> $x {
    $x *= 2;
}
# Now @arr = 2,4,6;
```

Ruby

```
set.each do |item|
  # do something to item
end
```

or

```
for item in set
  # do something to item
end
```

This can also be used with a hash.

```
set.each do |item,value|
  # do something to item
  # do something to value
end
```

Rust

The `for` loop has the structure `for <pattern> in <expression> { /* optional statements */ }`. It implicitly calls the `IntoIterator::into_iter` (https://doc.rust-lang.org/std/iter/trait.IntoIterator.html) method on the expression, and uses the resulting value, which must implement the `Iterator` (https://doc.rust-lang.org/std/iter/trait.Iterator.html) trait. If the expression is itself an iterator, it is used directly by the `for` loop through an implementation of `IntoIterator` for all `Iterators` (https://doc.rust-lang.org/std/iter/trait.IntoIterator.html#impl-IntoIterator-25) that returns the iterator unchanged. The loop calls the `Iterator::next` method on the iterator before executing the loop body. If `Iterator::next` returns `Some(_)`, the value inside is assigned to the pattern and the loop body is executed; if it returns `None`, the loop is terminated.

```
let mut numbers = vec![1, 2, 3];

// Immutable reference:
for number in &numbers { // calls IntoIterator::into_iter(&numbers)
    println!("{}", number);
}

for square in numbers.iter().map(|x| x * x) { // numbers.iter().map(|x| x * x) implements
Iterator
    println!("{}", square);
}

// Mutable reference:
for number in &mut numbers { // calls IntoIterator::into_iter(&mut numbers)
    *number *= 2;
}

// prints "[2, 4, 6]":
println!("{:?}", numbers);

// Consumes the Vec and creates an Iterator:
for number in numbers { // calls IntoIterator::into_iter(numbers)
    // ...
}

// Errors with "borrow of moved value":
// println!("{:?}", numbers);
```

Scala

```
// return list of modified elements
items map { x => doSomething(x) }
items map multiplyByTwo

for {x <- items} yield doSomething(x)
for {x <- items} yield multiplyByTwo(x)

// return nothing, just perform action
items foreach { x => doSomething(x) }
items foreach println

for {x <- items} doSomething(x)
for {x <- items} println(x)
```

```
// pattern matching example in for-comprehension
for ((key, value) <- someMap) println(s"$key -> $value")
```

Scheme

```
(for-each do-something-with a-list)
```

`do-something-with` is a one-argument function.

Smalltalk

```
collection do: [:item| "do something to item" ]
```

Swift

Swift uses the `for…in` construct to iterate over members of a collection.[19]

```
for thing in someCollection {
    // do something with thing
}
```

The `for…in` loop is often used with the closed and half-open range constructs to iterate over the loop body a certain number of times.

```
for i in 0..<10 {
    // 0..<10 constructs a half-open range, so the loop body
    // is repeated for i = 0, i = 1, …, i = 9.
}

for i in 0...10 {
    // 0...10 constructs a closed range, so the loop body
    // is repeated for i = 0, i = 1, …, i = 9, i = 10.
}
```

SystemVerilog

SystemVerilog supports iteration over any vector or array type of any dimensionality using the `foreach` keyword.

A trivial example iterates over an array of integers:

code	prints

`int  array_1d[] = '{ 3, 2, 1, 0 };`
 	
`foreach array_1d[index]`
`  $display("array_1d[%0d]: %0d", index, array_1d[index]);`
	`array_1d[0]: 3`
	`array_1d[1]: 2`
	`array_1d[2]: 1`
	`array_1d[3]: 0`

A more complex example iterates over an associative array of arrays of integers:

code	prints

`int  array_2d[string][] = '{ "tens": '{ 10, 11 }, "twenties": '{ 20, 21 } };`
 	
`foreach array_2d[key,index]`
`  $display("array_2d[%s,%0d]: %0d", key, index, array_2d[key,index]);`
	`array_2d[tens,0]: 10`
	`array_2d[tens,1]: 11`
	`array_2d[twenties,0]: 20`
	`array_2d[twenties,1]: 21`

Tcl

Tcl uses foreach to iterate over lists. It is possible to specify more than one iterator variable, in which case they are assigned sequential values from the list.

code	prints

`foreach {i j} {1 2 3 4 5 6} {`
`    puts "$i $j"`
`}`
	`1 2`
	`3 4`
	`5 6`

It is also possible to iterate over more than one list simultaneously. In the following `i` assumes sequential values of the first list, `j` sequential values of the second list:

code	prints

`foreach i {1 2 3} j {a b c}  {`
`    puts "$i $j"`
`}`
	`1 a`
	`2 b`
	`3 c`

Visual Basic .NET

```
For Each item In enumerable
    ' Do something with item.
Next
```

or without type inference

```
For Each item As type In enumerable
    ' Do something with item.
Next
```

Windows

Conventional command processor

Invoke a hypothetical `frob` command three times, giving it a color name each time.

```
C:\>FOR %%a IN ( red green blue ) DO frob %%a
```

Windows PowerShell

```
foreach ($item in $set) {
    # Do something to $item
}
```

From a pipeline

```
$list | ForEach-Object {Write-Host $_}

# or using the aliases
$list | foreach {write $_}
$list | % {write $_}
```

XSLT

```
 <xsl:for-each select="set">
   <!-- do something for the elements in <set> -->
 </xsl:for-each>
```

[20]

See also

- Do while loop
- For loop
- While loop
- Map (higher-order function)

References

1. "D Programming Language `foreach` Statement Documentation" (http://www.digitalmars.com/d/statement.html#ForeachStatement). Digital Mars. Retrieved 2008-08-04.
2. "SWI-Prolog -- foreach/2" (https://www.swi-prolog.org/pldoc/man?predicate=foreach/2). *www.swi-prolog.org*. Retrieved 2020-02-10.
3. "Proposed ECMAScript 4th Edition – Language Overview" (https://www.ecma-international.org/activities/Languages/Language%20overview.pdf) (PDF). Retrieved 2020-02-21.
4. "for each..in" (https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/statements.html#for_each..in). Retrieved 2020-02-21.
5. "for..in" (https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/statements.html#for..in). Retrieved 2020-02-21.
6. "C++11 Features in Visual C++ 11 - Visual C++ Team Blog - Site Home - MSDN Blogs" (http://blogs.msdn.com/b/vcblog/archive/2011/09/12/10209291.aspx). Blogs.msdn.com. 2011-09-12. Retrieved 2013-08-04.
7. "Range-based for loop (since C++11)" (https://en.cppreference.com/w/cpp/language/range-for). en.cppreference.com. Retrieved 2018-12-03.

8. "std::for_each - cppreference" (http://en.cppreference.com/w/cpp/algorithm/for_each). en.cppreference.com. Retrieved 2017-09-30.
9. "Qt 4.2: Generic Containers" (https://web.archive.org/web/20151123090839/http://doc.qt.digia.com/4.2/containers.html#the-foreach-keyword). Doc.qt.digia.com. Archived from the original (http://doc.qt.digia.com/4.2/containers.html#the-foreach-keyword) on 2015-11-23. Retrieved 2013-08-04.
10. Eric Niebler (2013-01-31). "Chapter 9. Boost.Foreach - 1.53.0" (http://www.boost.org/doc/libs/1_53_0/doc/html/foreach.html). Boost.org. Retrieved 2013-08-04.
11. "Range Clause" (https://golang.org/ref/spec#RangeClause). *The Go Programming Language Specification*. The Go Programming Language. Retrieved October 20, 2013.
12. "Enhanced for Loop - This new language construct[...]" "Java Programming Language, Section: Enhancements in JDK 5" (http://java.sun.com/j2se/1.5.0/docs/guide/language/index.html). Sun Microsystems, Inc. 2004. Retrieved 2009-05-26.
13. "The For-Each Loop" "The For-Each Loop" (http://java.sun.com/j2se/1.5.0/docs/guide/language/foreach.html). Sun Microsystems, Inc. 2008. Retrieved 2009-05-10.
14. "Implementing this interface allows an object to be the target of the "foreach" statement." "Iterable (Java Platform SE 6)" (http://java.sun.com/javase/6/docs/api/java/lang/Iterable.html). Sun Microsystems, Inc. 2004. Retrieved 2009-05-12.
15. [1] (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach)
16. "Object.keys" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys). *Mozilla Developer Network*. Retrieved May 7, 2014.
17. "Lua Programming/Tables - Wikibooks, open books for an open world" (https://en.wikibooks.org/wiki/Lua_Programming/Tables#Foreach_loop). *en.wikibooks.org*. Retrieved 2017-12-06.
18. Chu, Oliver. "Mint Tutorial" (http://prezi.com/ougvv1wzx2lb/mint-tutorial-part-0/). Retrieved 20 October 2013.
19. https://developer.apple.com/library/prerelease/ios/documentation/swift/conceptual/swift_progran CH9-XID_153
20. "XSLT <xsl:for-each> Element" (https://www.w3schools.com/xsl/xsl_for_each.asp). *W3Schools.com*.
