

# **1 APR::Table - Perl API for manipulating APR opaque string-content tables**

## 1.1 Synopsis

```

use APR::Table ();

$table = APR::Table::make($pool, $nelts);
$table_copy = $table->copy($pool);

$table->clear();

$table->set($key => $val);
$table->unset($key);
$table->add($key, $val);

$val = $table->get($key);
@val = $table->get($key);

$table->merge($key => $val);

use APR::Const -compile qw(:table);
$table_overlay = $table_base->overlay($table_overlay, $pool);
$table_overlay->compress(APR::Const::OVERLAP_TABLES_MERGE);

$table_a->overlap($table_b, APR::Const::OVERLAP_TABLES_SET);

$table->do(sub {print "key $_[0], value $_[1]\n"}, @valid_keys);

#Tied Interface
$value = $table->{$key};
$table->{$key} = $value;
print "got it" if exists $table->{$key};

foreach my $key (keys %{$table}) {
    print "$key = $table->{$key}\n";
}

```

## 1.2 Description

`APR::Table` allows its users to manipulate opaque string-content tables.

On the C level the "opaque string-content" means: you can put in `'\0'`-terminated strings and whatever you put in your get out.

On the Perl level that means that we convert scalars into strings and store those strings. Any special information that was in the Perl scalar is not stored. So for example if a scalar was marked as `utf8`, `tainted` or `tied`, that information is not stored. When you get the data back as a Perl scalar you get only the string.

The table's structure is somewhat similar to the Perl's hash structure, but allows multiple values for the same key. An access to the records stored in the table always requires a key.

The key-value pairs are stored in the order they are added.

The keys are case-insensitive.

However as of the current implementation if more than value for the same key is requested, the whole table is linearly searched, which is very inefficient unless the table is very small.

APR::Table provides a TIE Interface.

See *apr/include/apr\_tables.h* in ASF's *apr* project for low level details.

## 1.3 API

APR::Table provides the following functions and/or methods:

### 1.3.1 *add*

Add data to a table, regardless of whether there is another element with the same key.

```
$table->add($key, $val);
```

- **obj: \$table (APR::Table object)**

The table to add to.

- **arg1: \$key (string)**

The key to use.

- **arg2: \$val (string)**

The value to add.

- **ret: no return value**
- **since: 2.0.00**

When adding data, this function makes a copy of both the key and the value.

### 1.3.2 *clear*

Delete all of the elements from a table.

```
$table->clear();
```

- **obj: \$table (APR::Table object)**

The table to clear.

- **ret: no return value**
- **since: 2.0.00**

## 1.3.3 *compress*

Eliminate redundant entries in a table by either overwriting or merging duplicates:

```
$table->compress($flags);
```

- **obj: \$table (APR::Table object)**

The table to compress.

- **arg1: \$flags (APR::Const constant)**

```
APR::Const::OVERLAP_TABLES_MERGE -- to merge
APR::Const::OVERLAP_TABLES_SET   -- to overwrite
```

- **ret: no return value**
- **since: 2.0.00**

Converts multi-valued keys in `$table` into single-valued keys. This function takes duplicate table entries and flattens them into a single entry. The flattening behavior is controlled by the (mandatory) `$flags` argument.

When `$flags == APR::Const::OVERLAP_TABLES_SET`, each key will be set to the last value seen for that key. For example, given key/value pairs 'foo => bar' and 'foo => baz', 'foo' would have a final value of 'baz' after compression -- the 'bar' value would be lost.

When `$flags == APR::Const::OVERLAP_TABLES_MERGE`, multiple values for the same key are flattened into a comma-separated list. Given key/value pairs 'foo => bar' and 'foo => baz', 'foo' would have a final value of 'bar, baz' after compression.

Access the constants via:

```
use APR::Const -compile qw(:table);
```

or an explicit:

```
use APR::Const -compile qw(OVERLAP_TABLES_SET OVERLAP_TABLES_MERGE);
```

`compress()` combined with `overlay()` does the same thing as `overlap()`.

Examples:

- **APR::Const::OVERLAP\_TABLES\_SET**

Start with table `$table`:

```
foo => "one"
foo => "two"
foo => "three"
bar => "beer"
```

which is done by:

```
use APR::Const    -compile => ':table';
my $table = APR::Table::make($r->pool, TABLE_SIZE);

$table->set(bar => 'beer');
$table->set(foo => 'one');
$table->add(foo => 'two');
$table->add(foo => 'three');
```

Now compress it using `APR::Const::OVERLAP_TABLES_SET`:

```
$table->compress(APR::Const::OVERLAP_TABLES_SET);
```

Now table `$table` contains:

```
foo => "three"
bar => "beer"
```

The value *three* for the key *foo*, that was added last, took over the other values.

- **APR::Const::OVERLAP\_TABLES\_MERGE**

Start with table `$table`:

```
foo => "one"
foo => "two"
foo => "three"
bar => "beer"
```

as in the previous example, now compress it using `APR::Const::OVERLAP_TABLES_MERGE`:

```
$table->compress(APR::Const::OVERLAP_TABLES_MERGE);
```

Now table `$table` contains:

```
foo => "one, two, three"
bar => "beer"
```

All the values for the same key were merged into one value.

## 1.3.4 copy

Create a new table and copy another table into it.

```
$table_copy = $table->copy($p);
```

- **obj: \$table (APR::Table object)**

The table to copy.

- **arg1: \$p ( APR::Pool object )**

The pool to allocate the new table out of.

- **ret: \$table\_copy ( APR::Table object )**

A copy of the table passed in.

- **since: 2.0.00**

### 1.3.5 do

Iterate over all the elements of the table, invoking provided subroutine for each element. The subroutine gets passed as argument, a key-value pair.

```
$table->do(sub {...}, @filter);
```

- **obj: \$table ( APR::Table object )**

The table to operate on.

- **arg1: \$sub ( CODE ref/string )**

A subroutine reference or name to be called on each item in the table. The subroutine can abort the iteration by returning 0 and should always return 1 otherwise.

- **opt arg3: @filter ( ARRAY )**

If passed, only keys matching one of the entries in `f@filter` will be processed.

- **ret: no return value**

- **since: 2.0.00**

Examples:

- This filter simply prints out the key/value pairs and counts how many pairs did it see.

```
use constant TABLE_SIZE => 20;
our $filter_count;
my $table = APR::Table::make($r->pool, TABLE_SIZE);

# populate the table with ascii data
for (1..TABLE_SIZE) {
    $table->set(chr($_+97), $_);
}

$filter_count = 0;
$table->do("my_filter");
print "Counted $filter_count elements";

sub my_filter {
    my ($key, $value) = @_;
```

```

    warn "$key => $value\n";
    $filter_count++;
    return 1;
}

```

Notice that `my_filter` always returns 1, ensuring that `do ( )` will pass all the key/value pairs.

- This filter is similar to the one from the previous example, but this time it decides to abort the filtering after seeing half of the table, by returning 0 when this happens.

```

sub my_filter {
    my ($key, $value) = @_;
    $filter_count++;
    return $filter_count == int(TABLE_SIZE)/2 ? 0 : 1;
}

```

## 1.3.6 get

Get the value(s) associated with a given key. After this call, the data is still in the table.

```

$val = $table->get($key);
@val = $table->get($key);

```

- **obj: \$table ( APR::Table object )**

The table to search for the key.

- **arg1: \$key ( string )**

The key to search for.

- **ret: \$val or @val**

In the scalar context the first matching value returned (the oldest in the table, if there is more than one value). If nothing matches `undef` is returned.

In the list context the whole table is traversed and all matching values are returned. An empty list is returned if nothing matches.

- **since: 2.0.00**

## 1.3.7 make

Make a new table.

```

$table = APR::Table::make($p, $nelts);

```

- **obj: \$p ( APR::Pool object )**

The pool to allocate the pool out of.

- **arg1: \$nelts ( integer )**

The number of elements in the initial table. At least 1 or more. If 0 is passed APR will still allocate 1.

- **ret: \$table ( APR::Table object )**

The new table.

- **since: 2.0.00**

This table can only store text data.

## 1.3.8 merge

Add data to a table by merging the value with data that has already been stored using ", " as a separator:

```
$table->merge($key, $val);
```

- **obj: \$table ( APR::Table object )**

The table to search for the data.

- **arg1: \$key ( string )**

The key to merge data for.

- **arg2: \$val ( string )**

The data to add.

- **ret: no return value**
- **since: 2.0.00**

If the key is not found, then this function acts like add( ).

If there is more than one value for the same key, only the first (the oldest) value gets merged.

Examples:

- Start with a pair:

```
merge => "1"
```

and merge "a" to the value:

```
$table->set( merge => '1');
$table->merge(merge => 'a');
$val = $table->get('merge');
```



Result:

```
$val == "1, a";
```

- Start with a multivalued pair:

```
merge => "1"
merge => "2"
```

and merge "a" to the first value;

```
$table->set( merge => '1');
$table->add( merge => '2');
$table->merge(merge => 'a');
@val = $table->get('merge');
```

Result:

```
$val[0] == "1, a";
$val[1] == "2";
```

Only the first value for the same key is affected.

- Have no entry and merge "a";

```
$table->merge(miss => 'a');
$val = $table->get('miss');
```

Result:

```
$val == "a";
```

## 1.3.9 overlap

For each key/value pair in `$table_b`, add the data to `$table_a`. The definition of `$flags` explains how `$flags` define the overlapping method.

```
$table_a->overlap($table_b, $flags);
```

- **obj: \$table\_a (APR::Table object)**

The table to add the data to.

- **arg1: \$table\_b (APR::Table object)**

The table to iterate over, adding its data to table `$table_a`

- **arg2: \$flags (integer)**

How to add the table to table `$table_a`.

When `$flags == APR::Const::OVERLAP_TABLES_SET`, if another element already exists with the same key, this will over-write the old data.

When `$flags == APR::Const::OVERLAP_TABLES_MERGE`, the key/value pair from `$table_b` is added, regardless of whether there is another element with the same key in `$table_a`.

- **ret: no return value**
- **since: 2.0.00**

Access the constants via:

```
use APR::Const -compile qw(:table);
```

or an explicit:

```
use APR::Const -compile qw(OVERLAP_TABLES_SET OVERLAP_TABLES_MERGE);
```

This function is highly optimized, and uses less memory and CPU cycles than a function that just loops through table `$table_b` calling other functions.

Conceptually, `overlap()` does this:

```
apr_array_header_t *barr = apr_table_elts(b);
apr_table_entry_t *belt = (apr_table_entry_t *)barr->elts;
int i;

for (i = 0; i < barr->nelts; ++i) {
    if (flags & APR_OVERLAP_TABLES_MERGE) {
        apr_table_mergen(a, belt[i].key, belt[i].val);
    }
    else {
        apr_table_setn(a, belt[i].key, belt[i].val);
    }
}
```

Except that it is more efficient (less space and cpu-time) especially when `$table_b` has many elements.

Notice the assumptions on the keys and values in `$table_b` -- they must be in an ancestor of `$table_a`'s pool. In practice `$table_b` and `$table_a` are usually from the same pool.

Examples:

- **APR::Const::OVERLAP\_TABLES\_SET**

Start with table `$base`:

```
foo => "one"
foo => "two"
bar => "beer"
```

and table \$add:

```
foo => "three"
```

which is done by:

```
use APR::Const    -compile => ':table';
my $base = APR::Table::make($r->pool, TABLE_SIZE);
my $add  = APR::Table::make($r->pool, TABLE_SIZE);

$base->set(bar => 'beer');
$base->set(foo => 'one');
$base->add(foo => 'two');

$add->set(foo => 'three');
```

Now overlap using `APR::Const::OVERLAP_TABLES_SET`:

```
$base->overlap($add, APR::Const::OVERLAP_TABLES_SET);
```

Now table \$add is unmodified and table \$base contains:

```
foo => "three"
bar => "beer"
```

The value from table add has overwritten all previous values for the same key both had (*foo*). This is the same as doing `overlay()` followed by `compress()` with `APR::Const::OVERLAP_TABLES_SET`.

- **APR::Const::OVERLAP\_TABLES\_MERGE**

Start with table \$base:

```
foo => "one"
foo => "two"
```

and table \$add:

```
foo => "three"
bar => "beer"
```

which is done by:

```
use APR::Const    -compile => ':table';
my $base = APR::Table::make($r->pool, TABLE_SIZE);
my $add  = APR::Table::make($r->pool, TABLE_SIZE);

$base->set(foo => 'one');
$base->add(foo => 'two');

$add->set(foo => 'three');
$add->set(bar => 'beer');
```

Now overlap using `APR::Const::OVERLAP_TABLES_MERGE`:

```
$base->overlap($add, APR::Const::OVERLAP_TABLES_MERGE);
```

Now table `$add` is unmodified and table `$base` contains:

```
foo => "one, two, three"
bar => "beer"
```

Values from both tables for the same key were merged into one value. This is the same as doing `overlay()` followed by `compress()` with `APR::Const::OVERLAP_TABLES_MERGE`.

## 1.3.10 overlay

Merge two tables into one new table. The resulting table may have more than one value for the same key.

```
$table = $table_base->overlay($table_overlay, $p);
```

- **obj: \$table\_base (APR::Table object)**

The table to add at the end of the new table.

- **arg1: \$table\_overlay (APR::Table object)**

The first table to put in the new table.

- **arg2: \$p (APR::Pool object)**

The pool to use for the new table.

- **ret: \$table (APR::Table object)**

A new table containing all of the data from the two passed in.

- **since: 2.0.00**

Examples:

- Start with table `$base`:

```
foo => "one"
foo => "two"
bar => "beer"
```

and table `$add`:

```
foo => "three"
```

which is done by:

```

use APR::Const    -compile => ':table';
my $base = APR::Table::make($r->pool, TABLE_SIZE);
my $add  = APR::Table::make($r->pool, TABLE_SIZE);

$base->set(bar => 'beer');
$base->set(foo => 'one');
$base->add(foo => 'two');

$add->set(foo => 'three');

```

Now overlay using `APR::Const::OVERLAP_TABLES_SET`:

```
my $overlay = $base->overlay($add, APR::Const::OVERLAP_TABLES_SET);
```

That resulted in a new table `$overlay` (tables `add` and `$base` are unmodified) which contains:

```

foo => "one"
foo => "two"
foo => "three"
bar => "beer"

```

## 1.3.11 set

Add a key/value pair to a table, if another element already exists with the same key, this will over-write the old data.

```
$table->set($key, $val);
```

- **obj: \$table (APR::Table object)**

The table to add the data to.

- **arg1: \$key (string)**

The key to use.

- **arg2: \$val (string)**

The value to add.

- **ret: no return value**
- **since: 2.0.00**

When adding data, this function makes a copy of both the key and the value.

## 1.3.12 unset

Remove data from the table.

```
$table->unset($key);
```

- **obj:** `$table` (**APR::Table** object)

The table to remove data from.

- **arg1:** `$key` (string)

The key of the data being removed.

- **ret:** no return value
- **since:** 2.0.00

## 1.4 TIE Interface

`APR::Table` also implements a tied interface, so you can work with the `$table` object as a hash reference.

The following tied-hash function are supported: `FETCH`, `STORE`, `DELETE`, `CLEAR`, `EXISTS`, `FIRSTKEY`, `NEXTKEY` and `DESTROY`.

Note regarding the use of `values()`. `APR::Table` can hold more than one key-value pair sharing the same key, so when using a table through the tied interface, the first entry found with the right key will be used, completely disregarding possible other entries with the same key. With Perl 5.8.0 and higher `values()` will correctly list values the corresponding to the list generated by `keys()`. That doesn't work with Perl 5.6. Therefore to portably iterate over the key-value pairs, use `each()` (which fully supports multivalued keys), or `APR::Table::do`.

### 1.4.1 EXISTS

```
$ret = $table->EXISTS($key);
```

- **obj:** `$table` (**APR::Table** object)
- **arg1:** `$key` (string)
- **ret:** `$ret` (integer)

true or false

- **since:** 2.0.00

### 1.4.2 CLEAR

```
$table->CLEAR();
```

- **obj:** `$table` (**APR::Table** object)
- **ret:** no return value
- **since:** 2.0.00

### 1.4.3 STORE

```
$table->STORE($key, $val);
```

- **obj:** `$table` (APR::Table object)
- **arg1:** `$key` (string)
- **arg2:** `$val` (string)
- **ret:** no return value
- **since:** 2.0.00

### 1.4.4 DELETE

```
$table->DELETE($key);
```

- **obj:** `$table` (APR::Table object)
- **arg1:** `$key` (string)
- **ret:** no return value
- **since:** 2.0.00

### 1.4.5 FETCH

```
$ret = $table->FETCH($key);
```

- **obj:** `$table` (APR::Table object)
- **arg1:** `$key` (string)
- **ret:** `$ret` (string)
- **since:** 2.0.00

When iterating through the table's entries with `each()`, `FETCH` will return the current value of a multi-valued key. For example:

```
$table->add("a" => 1);
$table->add("b" => 2);
$table->add("a" => 3);

($k, $v) = each %$table; # (a, 1)
print $table->{a};       # prints 1

($k, $v) = each %$table; # (b, 2)
print $table->{a};       # prints 1

($k, $v) = each %$table; # (a, 3)
print $table->{a};       # prints 3 !!!

($k, $v) = each %$table; # (undef, undef)
print $table->{a};       # prints 1
```

1.5 See Also

## **1.5 See Also**

mod\_perl 2.0 documentation.

## **1.6 Copyright**

mod\_perl 2.0 and its core modules are copyrighted under The Apache Software License, Version 2.0.

## **1.7 Authors**

The mod\_perl development team and numerous contributors.



## Table of Contents:

|        |   |    |
|--------|---|----|
| 1      | APR::Table - Perl API for manipulating APR opaque string-content tables | 1  |
| 1.1    | Synopsis  | 2  |
| 1.2    | Description   | 2  |
| 1.3    | API   | 3  |
| 1.3.1  | add   | 3  |
| 1.3.2  | clear   | 3  |
| 1.3.3  | compress  | 4  |
| 1.3.4  | copy  | 5  |
| 1.3.5  | do  | 6  |
| 1.3.6  | get   | 7  |
| 1.3.7  | make  | 7  |
| 1.3.8  | merge   | 8  |
| 1.3.9  | overlap   | 9  |
| 1.3.10 | overlay   | 12 |
| 1.3.11 | set   | 13 |
| 1.3.12 | unset   | 13 |
| 1.4    | TIE Interface   | 14 |
| 1.4.1  | EXISTS  | 14 |
| 1.4.2  | CLEAR   | 14 |
| 1.4.3  | STORE   | 15 |
| 1.4.4  | DELETE  | 15 |
| 1.4.5  | FETCH   | 15 |
| 1.5    | See Also  | 16 |
| 1.6    | Copyright   | 16 |
| 1.7    | Authors   | 16 |