

PL/pgSQL Arrays



16

Copyright

© Postgres Professional, 2017–2025

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Igor Gnatyuk

Translated by: Liudmila Mantrova, Alexander Meleshko, Elena Sharafutdinova

Photo by: Oleg Bartunov (Phu monastery, Bhrikuti summit, Nepal)

Use of Course Materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Arrays and Their Usage in PL/pgSQL

Loops over Array Elements

Functions with a Variable Number of Arguments
and Polymorphic Functions

Array Usage in Tables

Array Types

Array

a set of numbered elements of the same type
one-dimensional, multidimensional

Initialization

usage without an explicit declaration (`type_name[]`)
implicit declaration when creating a base type or a table (`__type_name`)

Usage

elements as scalar values
array slices
operations on arrays: comparison, inclusion, intersection, concatenation,
usage with ANY or ALL instead of a subquery, etc.

Just like a composite type (a record), an array is not a scalar; it consists of several elements of another type. But unlike in records, a) all these elements are of the same type, and b) they are accessed by an integer index, not by name (here the term *index* is used in the mathematical sense of the word, not in the sense of a database index).

An array type does not have to be explicitly declared. When any type (or a table) is created, a new array type named `__type` is also declared. To declare an array variable, all you need is to append square brackets to the name of the element type.

An array is a full-fledged SQL type: you can create table columns of this type, pass arrays as function arguments, and so on. Array elements can be used as regular scalar values. Array slices can also be used.

Arrays can be compared and checked for NULL; you can search arrays for element inclusion and intersection with other arrays, perform concatenation, etc. Arrays can also be applied in ANY/SOME and ALL constructs, similar to subqueries.

<https://postgrespro.com/docs/postgresql/16/arrays>

You can find various array functions in course handouts.

Initializing an Array and Referencing its Elements

Declaring a variable and initializing an array:

```
=> DO $$
DECLARE
    a integer[2]; -- the size is ignored
BEGIN
    a := ARRAY[10,20,30];
    RAISE NOTICE '%', a;
    -- by default, one-based indexing is used
    RAISE NOTICE 'a[1] = %, a[2] = %, a[3] = %', a[1], a[2], a[3];
    -- array slice
    RAISE NOTICE 'Slice [2:3] = %', a[2:3];
    -- assign values to the array slice
    a[2:3] := ARRAY[222,333];
    -- output the array
    RAISE NOTICE '%', a;
END;
$$ LANGUAGE plpgsql;

NOTICE: {10,20,30}
NOTICE: a[1] = 10, a[2] = 20, a[3] = 30
NOTICE: Slice [2:3] = {20,30}
NOTICE: {10,222,333}
DO
```

A one-dimensional array can be constructed element by element: it will be expanded automatically if required. If you omit some of the elements, they will receive NULL values.

What will be displayed?

```
=> DO $$
DECLARE
    a integer[];
BEGIN
    a[2] := 10;
    a[3] := 20;
    a[6] := 30;
    RAISE NOTICE '%', a;
END;
$$ LANGUAGE plpgsql;

NOTICE: [2:6]={10,20,NULL,NULL,30}
DO
```

Since element numbering begins with a value other than one, the array itself is preceded by a range of element indexes.

We can define a composite type and create an array of this type:

```
=> CREATE TYPE currency AS (amount numeric, code text);

CREATE TYPE

=> DO $$
DECLARE
    c currency[]; -- composite type array
BEGIN
    -- assign values to array elements
    c[1].amount := 10; c[1].code := 'RUB';
    c[2].amount := 50; c[2].code := 'KZT';
    RAISE NOTICE '%', c;
END
$$ LANGUAGE plpgsql;

NOTICE: {(10,RUB),"(50,KZT)"}
DO
```

Another way to construct an array is to use a subquery:

```
=> DO $$
DECLARE
    a integer[];
BEGIN
    a := ARRAY( SELECT n FROM generate_series(1,3) n );
    RAISE NOTICE '%', a;
END
$$ LANGUAGE plpgsql;
```

```
NOTICE: {1,2,3}
DO
```

You can also do the inverse operation: convert an array into a table:

```
=> SELECT unnest( ARRAY[1,2,3] );

unnest
-----
      1
      2
      3
(3 rows)
```

Fun fact: the IN clause with a list of values is transformed into a search operation over the array:

```
=> EXPLAIN (costs off)
SELECT * FROM generate_series(1,10) g(id) WHERE id IN (1,2,3);

          QUERY PLAN
-----
Function Scan on generate_series g
  Filter: (id = ANY ('{1,2,3}'::integer[]))
(2 rows)
```

A two-dimensional array is a rectangle matrix. Memory for the matrix is allocated at initialization. A literal looks like an array of arrays with the same number of elements. Here we have applied another initialization method that uses a string literal.

Once initialized, a multidimensional array cannot be expanded.

```
=> DO $$
DECLARE
    a integer[][] := '{
        { 10, 20, 30},
        {100,200,300}
    }';
BEGIN
    RAISE NOTICE 'Two-dimensional array : %', a;
    RAISE NOTICE 'Unlimited array slice [2:] = %', a[2:];
    -- assign values to the slice
    a[2:] := ARRAY[ARRAY[111, 222, 333]];
    -- output the whole array again
    RAISE NOTICE '%', a;
    -- cannot be expanded in any dimension
    a[4][4] := 1;
END
$$ LANGUAGE plpgsql;

NOTICE: Two-dimensional array : {{10,20,30},{100,200,300}}
NOTICE: Unlimited array slice [2:] = {{100,200,300}}
NOTICE: {{10,20,30},{111,222,333}}
ERROR: array subscript out of range
CONTEXT: PL/pgSQL function inline_code_block line 15 at assignment
```

A regular loop over element indexes

`array_lower`

`array_upper`

A FOREACH loop over array elements

this approach is easier, but it does not provide access to indexes

To iterate through array elements, you can simply set up an integer FOR loop using functions that return the minimum and the maximum index of the array.

But there is also a specialized loop: FOREACH. In this case, a loop variable iterates through the elements, not their indexes. That's why the variable must be of the same type as the array elements (as always, if the elements are records, you can replace a single composite variable with several scalar ones).

If a loop contains the SLICE clause, it will iterate through array slices. For example, the rows of a two-dimensional array will be treated as its one-dimensional slices.

<https://postgrespro.com/docs/postgresql/16/plpgsql-control-structures#PLPGSQL-FOREACH-ARRAY>

Arrays and Loops

You can set up a loop that iterates through index values of array elements. The second parameter of the `array_lower` and `array_upper` functions defines the array dimension (1 denotes a one-dimensional array).

```
=> DO $$
DECLARE
    a integer[] := ARRAY[10,20,30];
BEGIN
    FOR i IN array_lower(a,1)..array_upper(a,1) LOOP
        RAISE NOTICE 'a[%] = %', i, a[i];
    END LOOP;
END
$$ LANGUAGE plpgsql;

NOTICE: a[1] = 10
NOTICE: a[2] = 20
NOTICE: a[3] = 30
DO
```

If you do not need to know index values, it's easier to iterate directly through the elements:

```
=> DO $$
DECLARE
    a integer[] := ARRAY[10,20,30];
    x integer;
BEGIN
    FOREACH x IN ARRAY a LOOP
        RAISE NOTICE '%', x;
    END LOOP;
END
$$ LANGUAGE plpgsql;

NOTICE: 10
NOTICE: 20
NOTICE: 30
DO
```

Index iteration in a two-dimensional array:

```
=> DO $$
DECLARE
    -- double square brackets are optional
    a integer[] := ARRAY[
        ARRAY[ 10, 20, 30],
        ARRAY[100,200,300]
    ];
BEGIN
    FOR i IN array_lower(a,1)..array_upper(a,1) LOOP -- over rows
        FOR j IN array_lower(a,2)..array_upper(a,2) LOOP -- over columns
            RAISE NOTICE 'a[%][%] = %', i, j, a[i][j];
        END LOOP;
    END LOOP;
END
$$ LANGUAGE plpgsql;

NOTICE: a[1][1] = 10
NOTICE: a[1][2] = 20
NOTICE: a[1][3] = 30
NOTICE: a[2][1] = 100
NOTICE: a[2][2] = 200
NOTICE: a[2][3] = 300
DO
```

You do not need a nested loop to iterate through the elements of a two-dimensional array:

```

=> DO $$
DECLARE
    a integer[] := ARRAY[
        ARRAY[ 10, 20, 30],
        ARRAY[100,200,300]
    ];
    x integer;
BEGIN
    FOREACH x IN ARRAY a LOOP
        RAISE NOTICE '%', x;
    END LOOP;
END
$$ LANGUAGE plpgsql;

NOTICE:  10
NOTICE:  20
NOTICE:  30
NOTICE: 100
NOTICE: 200
NOTICE: 300
DO

```

You can also loop through slices, not just individual elements, in a similar way. The SLICE value must be an integer, no larger than the array size, and the variable to store the slices into must be an array itself. Here is a loop over one-dimensional slices as an example:

```

=> DO $$
DECLARE
    a integer[] := ARRAY[
        ARRAY[ 10, 20, 30],
        ARRAY[100,200,300]
    ];
    x integer[];
BEGIN
    FOREACH x SLICE 1 IN ARRAY a LOOP
        RAISE NOTICE '%', x;
    END LOOP;
END
$$ LANGUAGE plpgsql;

NOTICE:  {10,20,30}
NOTICE:  {100,200,300}
DO

```


Routines with a variable number of arguments

- all optional arguments must be of the same type
- optional arguments are passed to the routine as an array
- the last parameter array is defined by the VARIADIC mode

Polymorphic routines

- support arguments of various types;
- the actual type is defined at run time
- can use additional polymorphic pseudotypes *anyarray*, *anynonarray*, *anycompatiblearray* and *anycompatiblenonarray*
- can have a variable number of arguments

Using arrays, you can create routines (functions and procedures) with a variable number of arguments.

While parameters with default values have to be explicitly specified in routine declaration, optional arguments can be passed with no limit: they are provided as an array. Consequently, all of them must be of the same type (or a compatible one, if *anycompatible* or *anycompatiblearray* is used).

The last parameter in routine declaration must be marked as VARIADIC; it must be of an array type.

<https://postgrespro.com/docs/postgresql/16/xfunc-sql#XFUNC-SQL-VARIADIC-FUNCTIONS>

We have already mentioned polymorphic routines which can accept arguments of various types. Routine declaration uses a special polymorphic pseudotype, while the actual type is defined at run time based on the types of the passed arguments.

There are special polymorphic types *anyarray* and *anycompatiblearray* (and *anynonarray* and *anycompatiblenonarray* for non-arrays).

These types can be used when passing a variable number of arguments via a VARIADIC parameter.

<https://postgrespro.com/docs/postgresql/16/xfunc-sql#XFUNC-SQL-POLYMORPHIC-FUNCTIONS>

Arrays and Routines

When discussing polymorphism and overloading as part of the SQL. Procedures lecture, we created the maximum function to compare three numbers and find the greatest one. Now let's generalize this function, so that it can be used with an arbitrary number of arguments. For this purpose, we'll declare a single VARIADIC parameter:

```
=> CREATE FUNCTION maximum(VARIADIC a integer[]) RETURNS integer
AS $$
DECLARE
    x integer;
    maxsofar integer;
BEGIN
    FOREACH x IN ARRAY a LOOP
        IF x IS NOT NULL AND (maxsofar IS NULL OR x > maxsofar) THEN
            maxsofar := x;
        END IF;
    END LOOP;
    RETURN maxsofar;
END
$$ IMMUTABLE LANGUAGE plpgsql;

CREATE FUNCTION
```

Let's try it out:

```
=> SELECT maximum(12, 65, 47);

 maximum
-----
        65
(1 row)
```

```
=> SELECT maximum(12, 65, 47, null, 87, 24);

 maximum
-----
        87
(1 row)
```

```
=> SELECT maximum(null, null);

 maximum
-----

(1 row)
```

To complete this illustration, we can make this function polymorphic as well, so that it takes any data type (which supports comparison operators, of course).

```
=> DROP FUNCTION maximum(integer[]);

DROP FUNCTION
```

- Polymorphic types anycompatiblearray and anycompatible (also anyarray and anyelement) must match each other: anycompatiblearray = anycompatible[], anyarray = anyelement[];
- We need a variable of the same type as an array element. But it cannot be declared as anycompatible: it must have an actual type. The %TYPE construct helps us out here.

```
=> CREATE FUNCTION maximum(VARIADIC a anycompatiblearray, maxsofar OUT anycompatible)
AS $$
DECLARE
    x maxsofar%TYPE;
BEGIN
    FOREACH x IN ARRAY a LOOP
        IF x IS NOT NULL AND (maxsofar IS NULL OR x > maxsofar) THEN
            maxsofar := x;
        END IF;
    END LOOP;
END
$$ IMMUTABLE LANGUAGE plpgsql;

CREATE FUNCTION
```

Let's check the result:

```
=> SELECT maximum(12, 65, 47);
```

```
maximum
-----
      65
(1 row)
```

```
=> SELECT maximum(12.1, 65.3, 47.6);
```

```
maximum
-----
     65.3
(1 row)
```

```
=> SELECT maximum(12, 65.3, 15e2, 3.14);
```

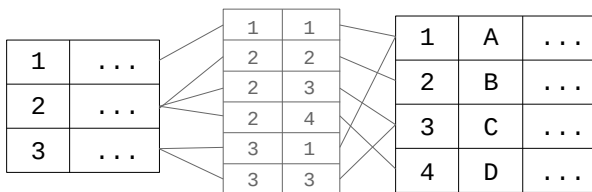
```
maximum
-----
    1500
(1 row)
```

Now our function is almost completely analogous to the greatest expression provided in SQL.

An Array or a Table?

1	...	{A}
2	...	{B, C, D}
3	...	{A, C}

compact representation
no joins required
convenient in simple cases



separate tables:
many to many relationship
a universal solution

A traditional relational approach assumes that a table stores atomic values (first normal form). The SQL language has no tools for peeking into composite values.

That's why a traditional approach relies on creating an additional table connected to the main one by a many-to-many relationship.

Nevertheless, we can create a table with a column of an *array* type. PostgreSQL offers a rich set of array functions; the search for an array element can be sped up using special indexes (covered in the DEV2 course).

This approach can be convenient: we get a concise representation that does not require any joins. For example, arrays are extensively used in PostgreSQL system catalog.

The choice of approach depends on the goals and the operations required. Consider the example in the demo.

An Array or a Table?

Imagine that we are designing a database for writing a blog. The blog contains some posts, and we would like to tag them.

The traditional approach is to create a separate table for tags. For example:

```
=> CREATE TABLE posts(  
    post_id integer PRIMARY KEY,  
    message text  
);
```

CREATE TABLE

```
=> CREATE TABLE tags(  
    tag_id integer PRIMARY KEY,  
    name text  
);
```

CREATE TABLE

Let's connect posts and tags by a many-to-many relationship via an additional table:

```
=> CREATE TABLE posts_tags(  
    post_id integer REFERENCES posts(post_id),  
    tag_id integer REFERENCES tags(tag_id)  
);
```

CREATE TABLE

Let's fill our tables with text data:

```
=> INSERT INTO posts(post_id,message) VALUES  
    (1, 'I set my password to "incorrect".'),  
    (2, 'Your future is whatever you make it, so make it a good one.');
```

INSERT 0 2

```
=> INSERT INTO tags(tag_id,name) VALUES  
    (1, 'my past and thoughts'), (2, 'technology'), (3, 'family');
```

INSERT 0 3

```
=> INSERT INTO posts_tags(post_id,tag_id) VALUES  
    (1,1), (1,2), (2,1), (2,3);
```

INSERT 0 4

Now we can display posts and tags:

```
=> SELECT p.message, t.name  
FROM posts p  
    JOIN posts_tags pt ON pt.post_id = p.post_id  
    JOIN tags t ON t.tag_id = pt.tag_id  
ORDER BY p.post_id, t.name \gx
```

```
-[ RECORD 1 ]-----  
message | I set my password to "incorrect".  
name    | my past and thoughts  
-[ RECORD 2 ]-----  
message | I set my password to "incorrect".  
name    | technology  
-[ RECORD 3 ]-----  
message | Your future is whatever you make it, so make it a good one.  
name    | family  
-[ RECORD 4 ]-----  
message | Your future is whatever you make it, so make it a good one.  
name    | my past and thoughts
```

Or we can do it a bit differently to get an array of tags. We are going to use an aggregate function for this purpose:

```
=> SELECT p.message, array_agg(t.name ORDER BY t.name) tags  
FROM posts p  
    JOIN posts_tags pt ON pt.post_id = p.post_id  
    JOIN tags t ON t.tag_id = pt.tag_id  
GROUP BY p.post_id  
ORDER BY p.post_id \gx
```

```

-[ RECORD 1 ]-----
message | I set my password to "incorrect".
tags    | {"my past and thoughts",technology}
-[ RECORD 2 ]-----
message | Your future is whatever you make it, so make it a good one.
tags    | {family,"my past and thoughts"}

```

We can find all posts with a particular tag:

```

=> SELECT p.message
FROM posts p
     JOIN posts_tags pt ON pt.post_id = p.post_id
     JOIN tags t ON t.tag_id = pt.tag_id
WHERE t.name = 'my past and thoughts'
ORDER BY p.post_id;

          message
-----
I set my password to "incorrect".
Your future is whatever you make it, so make it a good one.
(2 rows)

```

We may also need to find all the unique tags, and it is really easy:

```

=> SELECT t.name
FROM tags t
ORDER BY t.name;

          name
-----
family
my past and thoughts
technology
(3 rows)

```

Now let's try another approach to this task. Suppose the tags are stored as a text array right inside the table with posts.

```

=> DROP TABLE posts_tags;

DROP TABLE

=> DROP TABLE tags;

DROP TABLE

=> ALTER TABLE posts ADD COLUMN tags text[];

ALTER TABLE

```

There are no tag IDs, but we don't really need them.

```

=> UPDATE posts SET tags = '{"my past and thoughts","technology"}'
WHERE post_id = 1;

UPDATE 1

=> UPDATE posts SET tags = '{"my past and thoughts","family"}'
WHERE post_id = 2;

UPDATE 1

```

Now it's easier to display all posts:

```

=> SELECT p.message, p.tags
FROM posts p
ORDER BY p.post_id \gx

-[ RECORD 1 ]-----
message | I set my password to "incorrect".
tags    | {"my past and thoughts",technology}
-[ RECORD 2 ]-----
message | Your future is whatever you make it, so make it a good one.
tags    | {"my past and thoughts",family}

```

It is also easy to find all posts with the same tag (using the intersection operator &&).

This operation can be sped up using GIN index, and the query won't require searching through the whole table of posts.

```
=> SELECT p.message
FROM posts p

WHERE p.tags && '{"my past and thoughts"}'
ORDER BY p.post_id;
```

```

              message
-----
I set my password to "incorrect".
Your future is whatever you make it, so make it a good one.
(2 rows)
```

But it is quite hard to get the list of all tags. It requires unnesting all the tag arrays into a big table, and the search for unique values in this table is quite resource-intensive.

```
=> SELECT DISTINCT unnest(p.tags) AS name
FROM posts p;
```

```

              name
-----
family
technology
my past and thoughts
(3 rows)
```

We can clearly see data duplication here.

Thus, both approaches have the right to be applied.

In simple cases, arrays look more straightforward and work well.

In more complex scenarios (imagine that we would like to store the date of tag creation together with its name, or we need to use check constraints), the traditional approach becomes more preferable.

An array consists of numbered elements of the same data type

An array column is an alternative to a separate table: it offers convenient operations on arrays and index support

Arrays enable you to create functions with a variable number of arguments

1. Create a function `add_book` for adding a new book.
The function must take two arguments: the name of the book and an array of author IDs. It must return the ID of the added book.
Check that the application now allows adding books.

1.

```
FUNCTION add_book(title text, authors integer[])  
RETURNS integer
```

1. The add_book Function

```
=> CREATE FUNCTION add_book(title text, authors integer[])
RETURNS integer
AS $$
DECLARE
    book_id integer;
    id integer;
    seq_num integer := 1;
BEGIN
    INSERT INTO books(title)
    VALUES(title)
    RETURNING books.book_id INTO book_id;
    FOREACH id IN ARRAY authors LOOP
        INSERT INTO authorship(book_id, author_id, seq_num)
        VALUES (book_id, id, seq_num);
        seq_num := seq_num + 1;
    END LOOP;
    RETURN book_id;
END
$$ VOLATILE LANGUAGE plpgsql;

CREATE FUNCTION
```

1. Implement a function *map* that takes two arguments: array of *real* numbers and the name of an additional function that takes a single parameter of the *real* type.
The *map* function must apply the additional function to each element of the array and return the result.
2. Implement a function *reduce* that takes two arguments: an array of *real* numbers and the name of an additional function that takes two arguments of the *real* type.
The function must return a *real* number calculated by sequentially folding the array from left to right.
3. Make the functions *map* and *reduce* polymorphic.

1. For example:

```
map(ARRAY[4.0,9.0], 'sqrt') → ARRAY[2.0,3.0]
```

2. For example:

```
reduce(ARRAY[1.0,3.0,2.0,0.5], 'greatest') → 3.0
```

In this case, the value is calculated as follows:

```
greatest( greatest( greatest(1.0,3.0), 2.0 ), 0.5 )
```

1. The map Function

```
=> CREATE DATABASE plpgsql_arrays;
```

```
CREATE DATABASE
```

```
=> \c plpgsql_arrays
```

You are now connected to database "plpgsql_arrays" as user "student".

```
=> CREATE FUNCTION map(a INOUT float[], func text)
AS $$
DECLARE
    i integer;
    x float;
BEGIN
    IF cardinality(a) > 0 THEN
        FOR i IN array_lower(a,1)..array_upper(a,1) LOOP
            EXECUTE format('SELECT %I($1)', func) USING a[i] INTO x;
            a[i] := x;
        END LOOP;
    END IF;
END
$$ IMMUTABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

- INTO a[i] doesn't work, need a separate variable.

```
=> SELECT map(ARRAY[4.0,9.0,16.0], 'sqrt');
```

```
map
-----
{2,3,4}
(1 row)
```

```
=> SELECT map(ARRAY[]::float[], 'sqrt');
```

```
map
-----
{}
(1 row)
```

Another implementation with a FOREACH loop:

```
=> CREATE OR REPLACE FUNCTION map(a float[], func text) RETURNS float[]
AS $$
DECLARE
    x float;
    b float[]; -- empty array
BEGIN
    FOREACH x IN ARRAY a LOOP
        EXECUTE format('SELECT %I($1)', func) USING x INTO x;
        b := b || x;
    END LOOP;
    RETURN b;
END
$$ IMMUTABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> SELECT map(ARRAY[4.0,9.0,16.0], 'sqrt');
```

```
map
-----
{2,3,4}
(1 row)
```

```
=> SELECT map(ARRAY[]::float[], 'sqrt');
```

```
map
-----
(1 row)
```

2. The reduce Function

```
=> CREATE FUNCTION reduce(a float[], func text) RETURNS float
AS $$
DECLARE
    i integer;
    r float := NULL;
BEGIN
    IF cardinality(a) > 0 THEN
        r := a[array_lower(a,1)];
        FOR i IN array_lower(a,1)+1 .. array_upper(a,1) LOOP
            EXECUTE format('SELECT %I($1,$2)',func) USING r, a[i]
                INTO r;
        END LOOP;
    END IF;
    RETURN r;
END
$$ IMMUTABLE LANGUAGE plpgsql;
```

CREATE FUNCTION

greatest (as well as least) is not a function but a built-in conditional expression, and cannot be used directly due to escaping:

```
=> SELECT reduce( ARRAY[1.0,3.0,2.0], 'greatest');
```

ERROR: function greatest(double precision, double precision) does not exist
LINE 1: SELECT "greatest"(\$1,\$2)
 ^

HINT: No function matches the given name and argument types. You might need to add explicit type casts.

QUERY: SELECT "greatest"(\$1,\$2)

CONTEXT: PL/pgSQL function reduce(double precision[],text) line 9 at EXECUTE

Instead, use the maximum function implemented in the demo.

```
=> CREATE FUNCTION maximum(VARIADIC a anycompatiblearray, maxsofar OUT anycompatible)
AS $$
DECLARE
    x maxsofar%TYPE;
BEGIN
    FOREACH x IN ARRAY a LOOP
        IF x IS NOT NULL AND (maxsofar IS NULL OR x > maxsofar) THEN
            maxsofar := x;
        END IF;
    END LOOP;
END
$$ IMMUTABLE LANGUAGE plpgsql;
```

CREATE FUNCTION

```
=> SELECT reduce(ARRAY[1.0,3.0,2.0], 'maximum');
```

```
reduce
-----
      3
(1 row)
```

```
=> SELECT reduce(ARRAY[1.0], 'maximum');
```

```
reduce
-----
      1
(1 row)
```

```
=> SELECT reduce(ARRAY[]::float[], 'maximum');
```

```
reduce
-----
(1 row)
```

The FOREACH option:

```

=> CREATE OR REPLACE FUNCTION reduce(a float[], func text) RETURNS float
AS $$
DECLARE
    x float;
    r float;
    first boolean := true;

BEGIN
    FOREACH x IN ARRAY a LOOP
        IF first THEN
            r := x;
            first := false;
        ELSE
            EXECUTE format('SELECT %I($1,$2)',func) USING r, x INTO r;
        END IF;
    END LOOP;
    RETURN r;
END
$$ IMMUTABLE LANGUAGE plpgsql;

CREATE FUNCTION

=> SELECT reduce(ARRAY[1.0,3.0,2.0], 'maximum');

    reduce
    -----
           3
(1 row)

=> SELECT reduce(ARRAY[1.0], 'maximum');

    reduce
    -----
           1
(1 row)

=> SELECT reduce(ARRAY[]::float[], 'maximum');

    reduce
    -----

(1 row)

```

3. Polymorphic Function Variants

The map function.

```

=> DROP FUNCTION map(float[],text);

DROP FUNCTION

=> CREATE FUNCTION map(
    a anyarray,
    func text,
    elem anyelement DEFAULT NULL
)
RETURNS anyarray
AS $$
DECLARE
    x elem%TYPE;
    b a%TYPE;
BEGIN
    FOREACH x IN ARRAY a LOOP
        EXECUTE format('SELECT %I($1)',func) USING x INTO x;
        b := b || x;
    END LOOP;
    RETURN b;
END
$$ IMMUTABLE LANGUAGE plpgsql;

CREATE FUNCTION



- A dummy parameter of the anyelement type is required to declare a variable of the same type inside the function.



=> SELECT map(ARRAY[4.0,9.0,16.0], 'sqrt');

```

```

map
-----
{2.000000000000000,3.000000000000000,4.000000000000000}
(1 row)

=> SELECT map(ARRAY[]::float[], 'sqrt');

map
-----

(1 row)

```

Example with a different data type:

```

=> SELECT map(ARRAY[' a ', ' b', 'c '], 'btrim');

map
-----
{a,b,c}
(1 row)

```

The reduce function.

```

=> DROP FUNCTION reduce(float[], text);

DROP FUNCTION

=> CREATE FUNCTION reduce(
    a anyarray,
    func text,
    elem anyelement DEFAULT NULL
)
RETURNS anyelement
AS $$
DECLARE
    x elem%TYPE;
    r elem%TYPE;
    first boolean := true;
BEGIN
    FOREACH x IN ARRAY a LOOP
        IF first THEN
            r := x;
            first := false;
        ELSE
            EXECUTE format('SELECT %I($1,$2)', func) USING r, x INTO r;
        END IF;
    END LOOP;
    RETURN r;
END
$$ IMMUTABLE LANGUAGE plpgsql;

CREATE FUNCTION

=> CREATE FUNCTION add(x anyelement, y anyelement) RETURNS anyelement
AS $$
BEGIN
    RETURN x + y;
END
$$ IMMUTABLE LANGUAGE plpgsql;

CREATE FUNCTION

=> SELECT reduce(ARRAY[1,-2,4], 'add');

reduce
-----
3
(1 row)

=> SELECT reduce(ARRAY['a','b','c'], 'concat');

reduce
-----
abc
(1 row)

=> \c postgres

```

You are now connected to database "postgres" as user "student".

```
=> DROP DATABASE plpgsql_arrays;
```

```
DROP DATABASE
```