

Arrays, Associative Arrays, and Strings

The previous chapters indirectly acquainted us with arrays, associative arrays, and strings—an expression here, a literal there—so it’s time for a closer inspection. A lot of good code can be written using only these three types, so learning about them comes just handy now that we have expressions and statements under our belt.

4.1 Dynamic Arrays

D offers a simple but very versatile array abstraction. For a type `T`, `T[]` is the type of contiguous regions of memory containing elements of type `T`. D calls `T[]` “array of values of type `T`” or, colloquially, “array of `T`s.”

To create a dynamic array, use a `new` expression (§ 2.4.12 on page 61) as follows:

```
15 int[] array = new int[20]; // create an array of 20 integers
```

or, simpler and more convenient:

```
auto array = new int[20]; // create an array of 20 integers
```

20 All elements of a freshly-created array of type `T[]` are initialized with `T.init`, which is `0` for integers. After creation, the array’s elements are accessible through the index expression `array[n]`:

```
25 auto array = new int[20];  
auto x = array[5]; // valid indices are 0 through 19
```

```

assert(x == 0);           // initial element values are int.init = 0
array[7] = 42;           // elements are assignable
assert(array[7] == 42);

```

- 5 The number of elements passed to the `new`-expression does not need to be constant. For example, the program below creates an array of random length and then fills it with random numbers, for which generation it enlists the help of the function `uniform` in module `std.random`:

```

10 import std.random;

void main() {
    auto array = new double[uniform(1, 128)];
    foreach (i; 0 .. array.length) {
15     array[i] = uniform(0.0, 1.0);
    }
    ...
}

```

- 20 The `foreach` loop above could be rewritten to refer directly to each array element instead of using indexing (recall § 3.8.5 on page 82):

```

foreach (ref element; array) {
    element = uniform(0.0, 1.0);
}

```

- 25 The `ref` informs the compiler that we want to reflect assignments to `element` back into the original array. Otherwise, `element` would be a copy of each array element in turn.

If you want to initialize an array with specific contents, you may want to use an array literal.

```

30 auto somePrimes = [2, 3, 5, 7, 11, 13, 17];

```

[Note: normally the code below would not be included. It is visible because it fails to corroborate statements made in the text.]

```

35 auto somePrimes = [2, 3, 5, 7, 11, 13, 17];
assert(is(typeof(somePrimes) == int[]));

```

- 40 Another way to create an array is by duplicating an existing one. The property `array.dup` yields an element-by-element copy of `array`.

```

auto array = new int[100];
...
auto copy = array.dup;

```

```

assert(array !is copy);           // the arrays are distinct...
assert(array == copy);           // ...but have equal contents

```

5 Finally, if you just define a variable of type `T[]` without initializing it or by initializing it with `null`, that's a null array. A `null` array has no elements and compares equal with `null`.

```

string[] a;           // same as string[] a = null
assert(a is null);
10 assert(a == null); // same as above
a = new string[2];
assert(a !is null);

```

4.1.1 Length

15 Dynamic arrays remember their length. To access it, use the array's `.length` property:

```

auto array = new short[55];
assert(array.length == 55);

```

20 The expression `array.length` occurs frequently inside an index expression for array. For example, the last element of array is `array[array.length - 1]`. To simplify such cases, the symbol '\$' inside an index expression stands for "the length of the array being indexed into."

```

25 auto array = new int[10];
array[9] = 42;
assert(array[$ - 1] == 42);

```

30 **[Note: normally the code below would not be included. It is visible because it fails to corroborate statements made in the text.]**

```

void main() {
    auto a = new int[2];
    // the name "length" should not pop up in an index expression
35 static assert(!is(typeof(a[length - 1])));
}

```

Effecting changes to an array's length is discussed in § ?? on page ??.

4.1.2 Bounds Checking

40 What happens if you do this?

```

auto array = new int[10];
auto invalid = array[100];

```

Given that arrays already know their own length, it is possible to insert the appropriate bounds checks, so feasibility is not an issue. The only problem is that bounds checking is one of the instances that painfully put efficiency and safety at odds.

For safety reasons, it is imperative to make sure, one way or another, that array accesses are within bounds. Out-of-bounds accesses may exhibit arbitrary behavior and expose the program to exploits and breakages.

However, thorough bounds checking still affects efficiency considerably with current compiler technology. Efficient bounds checking is the target of intensive research. One popular approach is to start with a fully-checked program and remove as many checks as a static analyzer can prove redundant. In the general case that becomes quickly difficult, in particular when uses of arrays cross procedure and module boundaries. Today's approaches require a long analysis time even for modest programs and remove only a fraction of checks [32].

D is in a conflicted position regarding the bounds checking conundrum. The language is trying to offer at the same time the safety and convenience of modern languages, and the ultimate unmuffled performance sought by system level programmers. The bounds checking issue implies a choice between the two, and D allows you to make that choice instead of making it for you.

D makes two distinctions during compilation:

- safe module vs. system module, and
- non-release build vs. release build.

D distinguishes between modules that are “safe” and modules that are “system.” The programmer decides how to categorize each module. When compiling a safe module, the compiler statically disables all language features that could cause memory corruption, including unchecked array indexing. When compiling a system module, the compiler allows raw, unchecked access to hardware. You may choose whether a given module is safe or system by using a command-line option or by using the `module` declaration at the beginning of your files like this:

[Note: the code below does not work due to a bug in the current compiler.]

```

module(safe) my_module_name;

```

vs.

```

module(system) my_module_name;

```

Chapter ?? explains in detail how module safety works, but at this point the important tidbit of information is that there are ways for you, the application developer, to choose whether a module you're working on is safe or system. Most, if not all, modules of a typical application should be safe.

Module safety is orthogonal to choosing a *release* build for your application. You direct the D compiler to build a release version by passing it a command-line flag (`-release` in the reference implementation). In a safe module, array bounds are *always* checked. In a system module, bounds checking is inserted only for non-release builds. In a non-release build, the compiler also inserts other checks such as `assert` expressions and contract assertions (see Ch. 8 for a thorough discussion on what the release mode entails). The interaction between safe vs. system modules and release vs. non-release modes is summarized in Table 4.1.

Table 4.1: Presence of bounds checking depending on module kind and build mode.

	Safe module	System module
Non-release build	✓	✓
Release build (<code>-release</code> flag on <code>dmd</code>)	✓	✗

[See Issue #3407 at http://d.puremagic.com/issues/show_bug.cgi?id=3407]

10 You've been warned.

4.1.3 Slicing

Slicing is a powerful feature that allows you to select and work with only a contiguous portion of an array. For example, say we want to print only the last half of an array.

```
15 import std.stdio;

void main() {
    auto array = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
    // Print only the last half
20    writeln(array[$ / 2 .. $]);
}
```

The program above prints

```
5 6 7 8 9
```

25 To extract a slice out of array, use the notation `array[m .. n]`, which extracts the portion of the array starting at index `m` and ending with (and including) index `n - 1`. The slice has the same type as array itself, so you can, for example, reassign the slice back to the array it originated from:

```
array = array[$ / 2 .. $];
```

30 The symbol '`$`' may participate in an expression inside either limit of the slice, and—just like in the case of simple indexing—stands in for the length of the array being sliced. The situation `m == n` is acceptable and yields an empty slice. However, slices with `m > n` or

$n > \text{array.length}$ are illegal. Checking for such illegal cases obeys the rules described just above (§ 4.1.2 on page 103).

The expression `array[0 .. $]` extracts a slice comprehending the entire contents of array. That expression is encountered very often, so the language gives a hand by making `array[]` equivalent to `array[0 .. $]`.

4.1.4 Copying

At a minimum, an array object keeps (or can compute in negligible time) two data key pieces of information, namely the upper and lower bounds of its data chunk. For example, executing

```
10 auto a = [1, 5, 2, 3, 6];
```

leads to a state illustrated in Fig. 4.1. The array only “sees” the region in between its bounds; the hashed area is inaccessible to it.

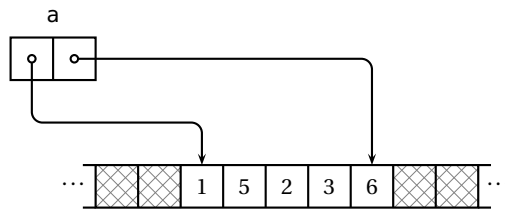


Figure 4.1: An array object referring to a chunk of 5 elements.

(Other representations are possible, for example storing the address of the first element and the length of the block, or the address of the first element and the address just past the last element. All representations have access to the same essential information.)

Initializing one array from another (`auto b = a;`) or assigning one array from another (`int[] b; ... b = a;`) does not automatically copy data under the hood. Such operations simply make `b` refer to the same memory chunk as `a`, as shown in Fig. 4.2 on the next page.

Furthermore, taking a slice off `b` reduces the chunk “seen” by `b`, again without copying it. Starting from the state in Fig. 4.2, if we now execute

```
b = b[1 .. $ - 2];
```

then `b` shrinks in range, again without any data copying (Fig. 4.3 on the facing page).

As a direct consequence of the data sharing illustrated above, writing an element of one array may be reflected in others:

```
int[] array = [0, 1, 2];
```

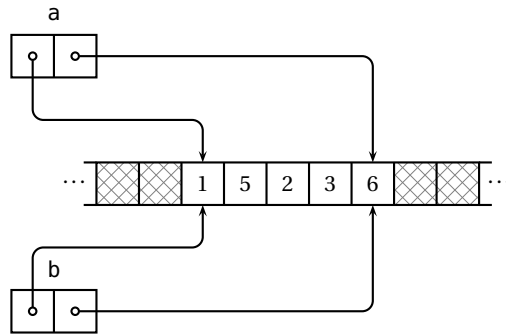


Figure 4.2: Executing “`auto b = a;`” does not copy the contents of `a`, but creates a new array object referring to the same chunk of data.

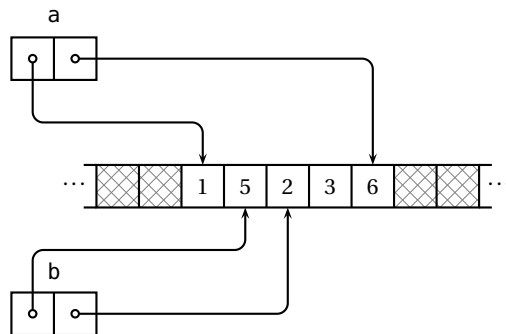


Figure 4.3: Executing “`b = b[1 .. $ - 2];`” shrinks the chunk controlled by `b` without copying the selected slice.

```

5 int[] slice = array[1 .. $];
assert(slice.length == 2);
slice[1] = 33;
assert(array[2] == 33); // Writing to slice affected array

```

4.1.5 Comparing for Equality

The expression `a is b` (§ 2.4.14.3 on page 64) compares the bounds of the two arrays for equality and yields `true` if and only if `a` and `b` are bound to the same exact region of memory. No comparison of content is carried at all.

To compare arrays `a` and `b` for element-for-element equality, use `a == b` or its negation `a != b` (§ 2.4.7 on page 56).

[Note: the code below does not work due to a bug in the current compiler.]

```

5 auto a = ["hello", "world"];
  auto b = a;
  assert(a is b);           // pass, a and b have the same bounds
  assert(a == b);         // pass, of course
  b = a.dup;
10 assert(a == b);         // pass, a and b are equal although
                          // they have different locations
  assert(a !is b);        // pass, a and b are different although
                          // they have equal contents

```

15 Comparing for equality iterates in lockstep through all elements of the two arrays and compares them in turn with `'=='`.

4.1.6 Concatenating

The construct

```
lhs ~ rhs
```

20 is a concatenation expression. The result of a concatenation is a new array with the contents of `lhs` followed by the contents of `rhs`. You may concatenate: (a) two arrays of types `T[]` and `T[]`; (b) array with value (`T[]` and `T`); and (c) value with array (`T` and `T[]`).

```

int[] a = [0, 10, 20];
25 int[] b = a ~ 42;
  assert(b == [0, 10, 20, 42]);
  a = b ~ a ~ 15;
  assert(a.length == 8);

```

30 A concatenation always creates a new array.

4.1.7 Array-wise Expressions

A few operations apply to arrays as a whole, without any explicit iteration. To create an array-wise expression, specify a trailing `'[]'` or `'[m .. n]'` on all slices involved in the expression, including the *left*-hand side of assignments, like this:

```

35 auto a = new double[4];           // must be already allocated
  auto a1 = [ 0.5, -0.5, 1.5, 2 ];
  auto a2 = [ 3.5, 5.5, 4.5, -1 ];
  a[] = (a1[] + a2[]) / 2;         // take the average of b and c

```

```
assert(a == [ 2.0, 2.5, 3.0, 0.5 ]);
```

An array-wise expression has one of the following forms:

- a single element, e.g., 5
- 5 • a slice explicitly trailed with '[' or '[m .. n]', e.g., a[] or a[1 .. \$ - 1]
- any valid D expression involving the two terms above, the unary operators '-', and '~', and the binary operators '+', '-', '*', '/', '%', '^', '&', '|', '=', '+=', '-=', '*=', '/=', '%=', '^=', '&=', and '|='.

The effect of an array expression is that of a loop assigning each element of the left-hand side in turn with the corresponding index of the right-hand side. For example, the assignment:

```
auto a = [1.0, 2.5, 3.6];
auto b = [4.5, 5.5, 1.4];
15 auto c = new double[3];
c[] += 4 * a[] + b[];
```

is the same as:

```
foreach (i; 0 .. c.length) {
20   c[i] += 4 * a[i] + b[i];
}
```

Bounds checking rules apply normally per § 4.1.2 on page 103.

Using slices suffixed with '[' or '[m .. n]', numbers, and the allowed operators, you may form parenthesized expressions of any depth and complexity, for example:

25 **[Note: the code below does not work due to a bug in the current compiler.]**

```
double[] a, b, c;
double d;
...
30 a[] = -(b[] * (c[] + 4)) + c[] * d;
```

One popular use of array-wise operations is simple filling and copying:

```
int[] a = new int[128];
35 int[] b = new int[128];
...
b[] = -1; // fill all of b with -1
a[] = b[]; // copy b's data over a's data
```

Warning Array-wise operations are powerful, but with great power comes great responsibility. You are responsible in making sure that the lvalue and the rvalue parts of any assignment in an array-wise operation do not overlap. The compiler is free to assume that when optimizing the operations into primitive vector operations offered by the host processor. If you do have overlapping, you'll need to write the loops by hand, in which case the compiler is not allowed to make any unchecked assumptions.

4.1.8 Shrinking

Array shrinking means that the array should “forget” about some elements from either the left or the right end, without needing to move the rest. The restriction on moving is important; if moving elements was an option, arrays would be easy to shrink—just create a new copy containing the elements to be kept.

Shrinking an array is the easiest thing: just assign to the array a slice of itself.

[Note: the code below does not work due to a bug in the current compiler.]

```

15 auto array = [0, 2, 4, 6, 8, 10];
   array = array[0 .. $ - 2];           // Right-shrink by two elements
   assert(array == [0, 2, 4, 6]);
   array = array[1 .. $];              // Left-shrink by one element
   assert(array == [2, 4, 6]);
20 array = array[1 .. $ - 1];          // Shrink from both sides
   assert(array == [4]);

```

All shrink operations take time independent of the array's length (practically only cost a couple of word assignments). Affordable shrinking from both ends is a very useful feature of D arrays. (Other languages allow cheap array shrinking from the right, but not from the left because the latter would involve moving over all elements of the array to preserve the location of the array's left edge.) In D you can take a copy of the array and progressively shrink it to systematically manipulate elements of the array, in confidence that the constant-time shrinking operations do not affect processing time by a large margin.

For example, let's write a little program that detects palindrome arrays passed via the command line. A palindrome array is left-right symmetric, e.g., [5, 17, 8, 17, 5] is palindrome, but [5, 7, 8, 7] is not. We need to avail ourselves of a few helpers. One is command line fetching, which nicely comes as an array of strings if you define main as main(string[] args). Then, we need to convert arguments from strings to ints, for which we use the function aptly named “to” in the std.conv module. For some string str, evaluating to!int(str) parses str into an int. Armed with these features, we can write the palindrome test program like this:

```

40 import std.conv, std.stdio;

```

```

void main(string[] args) {
    // Get rid of the program name
    args = args[1 .. $];
    while (args.length >= 2) {
5      if (to!int(args[0]) != to!int(args[$ - 1])) {
        writeln("not palindrome");
        return;
      }
      args = args[1 .. $ - 1];
10    }
    writeln("palindrome");
}

```

First, the program must get rid of the program name from the argument list, of which presence follows a tradition established by C. When you invoke our program (call it “palindrome”) like this:

```
palindrome 34 95 548
```

then the array `args` contains `["palindrome", "34", "95", "548"]`. Here’s where shrinking from the left `args = args[1 .. $]` comes in handy, reducing `args` to `["34", "95", "548"]`. Then, the program iteratively compares the two ends of the array. If they are different, there’s no purpose in continuing to test, so write “no palindrome” and bailout. If the test succeeds, `args` is reduced simultaneously from its left and right ends. Only if all tests succeed and `args` got shorter than two elements (the program considers arrays of zero or one elements palindromes), the program prints “palindrome” and exits. Although it does a fair amount of array manipulation, the program does not allocate any memory—it just starts with the preallocated array `args` and shrinks it.

4.1.9 Expanding

On to expanding arrays. To expand an array, use the append operator ‘`~=`’, for example:

[Note: the code below does not work due to a bug in the current compiler.]

```

30 auto a = [87, 40, 10];
   a ~= 42;
   assert(a == [87, 40, 10, 42]);
   a ~= [5, 17];
35 assert(a == [87, 40, 10, 42, 5, 17]);

```

Expanding arrays has a couple of subtleties with it that concern possible reallocation of the array. Consider:

[Note: the code below does not work due to a bug in the current compiler.]

```

auto a = [87, 40, 10, 2];
auto b = a;           // now a and b refer to the same chunk
a ~= [5, 17];        // append to a
5 a[0] = 15;          // modify a[0]
assert(b[0] == 15); // pass or fail?

```

Does the post-append assignment to `a[0]` also affect `b[0]`, or, in other words, do `a` and `b` still share data post-reallocation? The short answer is, `b[0]` may or may not be 15—the language makes no guarantee.

Realistically, there is no way to always have enough room at the end of `a` to reallocate it in place. At least sometimes, reallocation must occur. One easy way out would be to *always* reallocate `a` upon appending to it with `'~='`, thereby always making `a ~= b` the same exact thing as `a = a ~ b`, i.e. “allocate a new array consisting of `a` concatenated with `b` and then bind `a` to that new array.” Although that behavior is easiest to implement, it has serious efficiency problems. For example, oftentimes arrays are iteratively grown in a loop:

```

int[] a;
20 foreach (i; 0 .. 100) {
    a ~= i;
}

```

For 100 elements, pretty much any expansion scheme would work, but when arrays become larger, only solid solutions can remain reasonably fast. One particularly unsavory approach would be to allow the convenient but inefficient expansion syntax `a ~= b` for short arrays and discourage it on large arrays in favor of another, less convenient syntax. At best, the simplest and most intuitive syntax works for short and long arrays.

D leaves `'~='` the freedom of either expanding by reallocation or opportunistically expanding it in-place if there is enough unused memory at the end of the current array. The decision belongs entirely to the implementation of the `'~='`, but client code is guaranteed to benefit of good *average* performance over a large number of appends to the same array.

Fig. 4.4 on the facing page illustrates the two possible outcomes of the expansion request `a ~= [5, 17]`.

Depending on how the underlying memory allocator works, an array can expand in more ways than one.

- Often, allocators can only allocate chunks in specific sizes (e.g., powers of 2). It is therefore possible that a request for 700 bytes receives 1024 bytes of storage, of which 324 are slack. When an expansion request occurs, the array may check whether there's slack storage and use it.

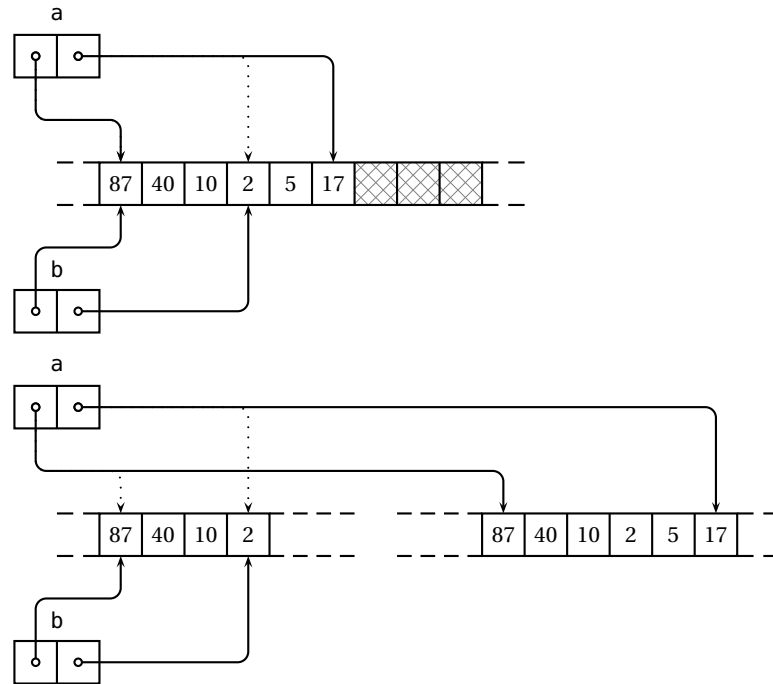


Figure 4.4: Two possible outcomes of an attempt to expand array *a*. In the first case (top), the memory chunk had available memory at its end, which is used for in-place expansion. In the second case, there was no more available room so a new chunk was allocated and *a* was adjusted to refer to it. Consequently, after expansion, *a*'s and *b*'s chunks may or may not overlap.

- If there is no slack space left, the array may initiate a more involved negotiation with the underlying memory allocator. “You know, I’m sitting here and could use some space to the right. Is by any chance the adjacent block available?” The allocator may find an empty block to the right of the current block and gobble it into its own block. This operation is known as *coalescing*. Then expansion can still proceed without moving any data.
- Finally, if there is absolutely no room in the current block, the array allocates a brand new block and moves all of its data in it. The array may deliberately allocate extra slack space when, for example, it detects repeated expansions of the same array.

An expanding array never stomps on an existing array. For example:

[Note: the code below does not work due to a bug in the current compiler.]

```

int[] a = [0, 10, 20, 30, 40, 50, 60, 70];
auto b = a[4 .. $];
5 a = a[0 .. 4];
  // At this point a and b are adjacent
  a ~= [0, 0, 0, 0];
  assert(b == [40, 50, 60, 70]); // passes; a got reallocated

```

10 The code above is carefully crafted to fool a into thinking it has room at its end: initially a received a larger size, and then b received the upper part of a and a got reduced to its lower part. Prior to appending to a, the arrays occupy adjacent chunks with a to the left of b. The post-append `assert`, however, confirms that a actually got reallocated, not expanded in place. The append operator only appends in place when it can prove there

15 is no other array to the right of the expanding one, and is always free to conservatively reallocate whenever the slightest suspicion is afoot.

4.1.10 Assigning to `.length`

Assigning to `array.length` allows you to either shrink or expand array, depending on the relation of the new length with the old length. For example:

```

20 int[] array;
  assert(array.length == 0);
  array.length = 1000;      // Grow
  assert(array.length == 1000);
25 array.length = 500;
  assert(array.length == 500); // Shrink

```

30 If the array grows as a result of assigning to `.length`, the added elements are initialized with `T.init`. The growth strategy and guarantees are identical to those of the append operator `'~='` (§ 4.1.9 on page 111).

If the array shrinks as a result of assigning to `.length`, D guarantees the array is not reallocated. Practically, if $n \leq a.length$, `a.length = n` is equivalent to `a = a[0 .. n]`. (However, that guarantee does not also imply that further expansions of the array will avoid reallocation.)

35 You may carry read-modify-write operations with `.length`, for example:

[Note: the code below does not work due to a bug in the current compiler.]

```

auto array = new int[10];
array.length += 1000;      // Grow
40 assert(array.length == 1010);
  array.length /= 10;
  assert(array.length == 101); // Shrink

```

Not much magic happens here; all the compiler does is to rewrite `array.length <op>= b` into `array.length = array.length <op> b`. There is some minor magic involved though (just a sleight of hand, really): `array` is only evaluated once in the rewritten expression, which is relevant if `array` is actually some elaborate expression.

5 4.2 Fixed-Size Arrays

D offers arrays of a size known during compilation, declared for example like this:

```
int[128] someInts;
```

10 For each type `T` and size `n`, the type `T[n]` is distinct from any other—for example, `uint[10]` is distinct from `uint[11]` and also from `int[10]`.

All fixed-size array values are allocated statically at the place of declaration. If the array value is defined globally, it goes in the data segment of the program. If allocated inside a function, the array will be allocated on the stack of that function upon the function call. (This means that defining very large arrays in functions may be dangerous.)

15 Upon creation, a fixed-size array `T[n]` value has all of its data initialized to `T.init`. For example:

```
int[3] a;
assert(a == [0, 0, 0]);
```

You can initialize a `T[n]` with a literal:

```
int[3] a = [1, 2, 3];
assert(a == [1, 2, 3]);
```

30 Beware, however: if you replace `int[3]` above with `auto`, `a`'s type will be deduced as `int[]`, not `int[3]`. Although it seems logical that the type of `[1, 2, 3]` should be `int[3]` which in a way is more “precise” than `int[]`, it turns out that dynamically-sized arrays are used much more often than fixed-size arrays, so insisting on fixed-size array literals would have been a usability impediment and a source of unpleasant surprises. Effectively, the use of literals would have prevented the gainful use of `auto`. As it is, array literals are `T[]` by default, and `T[n]` if you *ask* for that specific type and if `n` matches the number of values in the literal (as the code above shows).

35 If you initialize a fixed-size array of type `T[n]` with a single value of type `T`, the entire array will be filled with that value:

```
int[4] a = -1;
assert(a == [-1, -1, -1, -1]);
```

If you plan to leave the array uninitialized and fill it at runtime, just specify `void` as an initializer:

```
int[1024] a = void;
```

Such uninitialized arrays are particularly useful for large arrays that serve as temporary buffers. Don't forget they are inherently unsafe.

Accessing elements of fixed-size arrays is done by using the indexing operator `a[i]`, the same way as for dynamic arrays. Iteration is also virtually identical to dynamic arrays. For example, creating an array of 1024 random numbers would go like this:

```
10 import std.random;

void main() {
    double[1024] array;
    foreach (i; 0 .. array.length) {
15     array[i] = uniform(0.0, 1.0);
    }
    ...
}
```

The loop could use `ref` values to use array elements without indexing:

```
foreach (ref element; array) {
    element = uniform(0.0, 1.0);
}
```

4.2.1 Length

Obviously, fixed-size arrays are aware of their length because it's stuck in their very type. Unlike dynamic arrays' length, the `.length` property is read-only and a static constant. This means you can use `array.length` for fixed-size arrays whenever a compile-time constant is required, for example straight in the length of another fixed-sized array definition:

```
30 int quadrupeds[100];
int legs[4 * quadrupeds.length]; // fine, 400 legs
```

Inside an index expression for array `a`, `'$'` can be used in lieu of `a.length`, and is, again, a compile-time expression.

4.2.2 Bounds Checking

Bounds checking for fixed-sized arrays has an interesting twist. Whenever indexing is used with a compile-time expression, the compiler checks validity during compilation and refuses to compile in case of an out-of-bounds access. For example:

```

int[10] array;
array[15] = 5; // Error!
// Array index 15 is out of bounds a[0 .. 10]!

```

If the expression is a runtime value, compile-time bounds checking is done on a best-effort basis, and runtime checking follows the same protocol as bounds checking for dynamic arrays (§ 4.1.2 on page 103).

4.2.3 Slicing

Taking any slice off an array of type $T[n]$ yields an array of type $T[]$ without an intervening copy.

```

10 int[5] array = [40, 30, 20, 10, 0];
auto slice1 = array[2 .. $]; // slice1 has type int[]
assert(slice1 == [20, 10, 0]);
auto slice2 = array[]; // same as array[0 .. $]
15 assert(slice2 == array);

```

Compile-time bounds checking is carried against either or both bounds when they are compile-time constants.

If you take a slice with compile-time-known limits $T[a1 .. a2]$ off an array $T[n]$, and if you request an array of type $T[a2 - a1]$, the compiler grants the request. (The default type yielded, e.g. if you use `auto`, is still $T[]$.) For example:

```

int[10] a;
int[] b = a[1 .. 7]; // fine
25 auto c = a[1 .. 7]; // fine, c also has type int[]
int[6] d = a[1 .. 7]; // fine, a[1 .. 7] copied into d

```

4.2.4 Copying and Implicit Conversion

Unlike dynamic arrays, fixed-size arrays have value semantics. This means that copying arrays, passing arrays into functions, and returning arrays from functions—these all copy entire arrays. For example:

[Note: the code below does not work due to a bug in the current compiler.]

```

int[3] a = [1, 2, 3];
35 int[3] b = a;
a[1] = 42;
assert(b[1] == 1); // b is an independent copy of a
int[3] fun(int[3] x, int[3] y) {
// x and y are copies of the arguments

```

```

    x[0] = y[0] = 100;
    return x;
}
5 auto c = fun(a, b);           // c has type int[3]
  assert(c == [100, 42, 3]);
  assert(b == [1, 2, 3]);     // b is unaffected by fun

```

Passing entire arrays by value may be inefficient for large arrays, but it has many advantages. One advantage is that short arrays and pass-by-value are frequently used in high-performance computing. Another advantage is that pass-by-value has a simple cure—whenever you want reference semantics, just use `ref` or automatic conversion to `T[]` (see below). Finally, value semantics makes fixed-size arrays consistent with many other aspects of the language. (Historically, D had reference semantics for fixed-size arrays, which turned out to be a continuous source of contortions and special casing in client code.)

Arrays of type `T[n]` are implicitly convertible to arrays of type `T[]`. The dynamic array thusly obtained is not allocated anew—it simply latches onto the bounds of the source array. Therefore, the conversion is considered unsafe if the source array is stack-allocated. The implicit conversion makes it easy to pass fixed-size arrays of type `T[n]` to functions expecting `T[]`. However, if a function returns `T[n]`, that not automatically converted to `T[]`—it must be `T[n]`.

```

double[3] point = [0, 0, 0];
double[] test = point;           // fine
double[3] fun(double[] x) {
25   double[3] result;
    result[] = 2 * x[];           // array-wise operation
    return result;
}
30 auto r = fun(point);           // fine, r has type double[3]

```

You can duplicate a fixed-size array with the `.dup` property (§ 4.1 on page 101) but you don't get an object of type `T[n]` back, you get a dynamically-allocated array of type `T[]` that contains a copy of the fixed-size array. This behavior is sensible given that you otherwise don't need to duplicate a fixed-size array—to obtain a duplicate of `a`, just say `auto copy = a`. With `.dup`, you get to make a dynamic copy of a fixed-sized array.

35 4.2.5 Comparing for Equality

Fixed-sized arrays may be compared with `is` and `'=='`, just like dynamic arrays (§ 4.1.5 on page 107). You may also transparently mix fixed-size and dynamic-size arrays in comparisons.

```

40 int[4] fixed = [1, 2, 3, 4];

```

```

auto anotherFixed = fixed;
assert(anotherFixed !is fixed); // not the same (value semantics)
assert(anotherFixed == fixed); // same data
auto dynamic = fixed[]; // fetches the limits of fixed
5 assert(dynamic is fixed);
assert(dynamic == fixed); // obviously
dynamic = dynamic.dup; // create a copy
assert(dynamic !is fixed);
10 assert(dynamic == fixed);

```

4.2.6 Concatenating

Concatenation follows rules similar to those governing concatenation of dynamic arrays (§ 4.1.6). There is one important difference. If you *ask* for a fixed-size array, you get a fixed-size array. Otherwise, you get a newly-allocated dynamic array. For example:

```

15 double[2] a;
double[] b = a ~ 0.5; // concat double[2] with value, get double[]
auto c = a ~ 0.5; // same as above
double[3] d = a ~ 1.5; // fine, explicitly ask for fixed-size array
20 double[5] e = a ~ d; // fine, explicitly ask for fixed-size array

```

Whenever a fixed-array is requested as the result of the concatenating operator '~', there is no dynamic allocation—the result is statically allocated and the result of the concatenation is copied into it.

25 4.2.7 Array-wise Operations

Array-wise operations on static arrays work similarly to those for dynamic arrays (§ 4.1.7). Wherever possible, the compiler performs compile-time bounds checking for arrays bearing static lengths involved in an array-wise expression. You may mix fixed-sized and dynamic arrays in expressions.

30 4.3 Multidimensional Arrays

Since `T[]` is a dynamic array with elements of type `T` and `T[]` itself is a type, it's easy to infer that `T[][]` is an array of `T[]`s, or, put another way, an array of arrays of `T`s. Each element of the outer array is in turn an array offering the usual array primitives. Let's give `T[][]` a test drive.

```

35 auto array = new double[][][5]; // array of five arrays of double,
// each initially null

```

```

// Make a triangular matrix
foreach (i, ref e; array) {
  e = new double[array.length - i];
}

```

The shape of array defined above is triangular: the first row has 5 doubles, the second has 4, and so on to the fifth one (technically row 4), which has one element. Multidimensional arrays obtained by simply composing dynamic arrays are called “jagged arrays” because their rows may assume arbitrary lengths (as opposed to the somewhat expected straight right edge obtained when all rows have the same length). Fig. 4.5 illustrates array’s emplacement in memory.

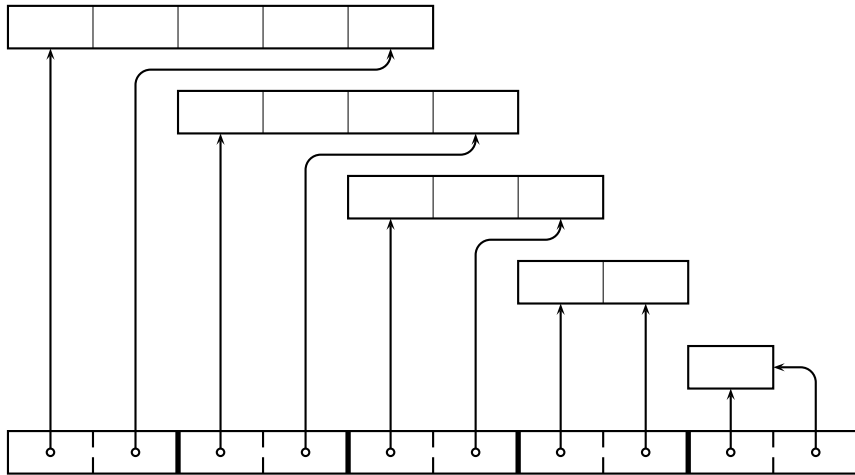


Figure 4.5: A jagged array storing a triangular matrix as defined in the example on the previous page.

To access an element in a jagged array, specify indices for each dimension in turn, for example `array[3][1]` accesses the second element in the fourth row of a jagged array.

One characteristic of jagged arrays is that they are not contiguous. On the plus side, this means that jagged arrays can spread themselves in memory and require smaller amounts of contiguous memory. The ability of storing rows of different lengths may save quite a bit of memory, too. On the minus side, “tall and thin” arrays with many rows and few columns incur a large size overhead as there’s one array to keep per column. For example, one array with 1,000,000 rows each having only 10 integers needs to hold an array of 2,000,000 words (one array per row) plus the management overhead of 1,000,000 small blocks, which, depending on the memory allocator implementation, may be considerable relative to the small per-row payload of 40 words.

Jagged arrays may have problems with efficiency of access and cache friendliness. Each element access requires two indirections, first through the outer array corresponding to the row, and then through the inner array corresponding to the column. Iterating row-wise is not much of a problem if you first fetch the row and then use it, but going

5 column-wise through a jagged array is a cache miss bonanza.

If the number of columns is known during compilation, you can easily compose a fixed-size array with a dynamic array:

```
enum size_t columns = 128;
// Allocate a matrix with 64 rows and 128 columns
auto matrix = new double[columns][64];
// No need to allocate each row - they already exist in-situ
foreach (ref row; matrix) {
    ... // use row of type double[columns]
15 }
```

In the example above it is crucial to use `ref` with `foreach`. Without `ref`, the value semantics of `double[columns]` (§ 4.2.4 on page 117) would create a copy of each row being iterated, which is likely to put a damp on the speed of your code.

20 If you know both the number of rows and columns during compilation, you may want to use a fixed-sized array of fixed-sized arrays, as follows:

```
enum size_t rows = 64, columns = 128;
// Allocate a matrix with 64 rows and 128 columns
25 double[columns][rows] matrix;
// No need to allocate the array at all - it's a value
foreach (ref row; matrix) {
    ... // use row of type double[columns]
30 }
```

To access an element at row `i` and column `j`, write `matrix[i][j]`. One small oddity is that the declaration specifies the sizes for each dimension in right-to-left order (i.e., `double[columns][rows]`), but when accessing elements, indices come in left-to-right order. This is because `[]` and `[n]` in types bind right-to-left, but in expressions they

35 bind left-to-right.

A variety of multidimensional array shapes can be created by composing fixed-size arrays and dynamic arrays. For example `int[5][][15]` is a three-dimensional array consisting of 15 arrays, each being a dynamically-allocated array of blocks of 5 `ints` each.

4.4 Associative Arrays

40 An array could be thought of as a function that maps positive integers (indices) to values of some arbitrary type (the data stored in the array). The function is defined only for in-

ters from zero to the array's length minus one and is entirely tabulated by the contents of the array.

Seen from that angle, associative arrays introduce a certain generalization of arrays. Instead of integers, an associative array may accept an (almost) arbitrary type as domain. For each value in the domain, it is possible to map a value of a different type—similarly to an array's slot. The storage method and associated algorithms are different than those of arrays, but, just like an array, an associative array offers fast storage and retrieval of a value given its key.

The type of an associative array is suggestively denoted as $V[K]$, where K is the key type and V is the associated value type. For example, let's create and initialize an associative array that maps strings to integers.

```
int[string] aa = [ "hello":42, "world":75 ];
```

An associative array!literal is a comma-separated list of terms of the form `key : value`, enclosed in square brackets. In the case above the literal is informative enough to make the explicit type of `aa` redundant, so it's more comfortable to write

```
auto aa = [ "hello":42, "world":75 ];
```

The type of an associative array!literal is always determined by the first key/value pair. It would have been easy to find the most fitting type over all key/value pairs passed, but there is a thing as a compiler too clever for its own good. When defining associative array literals, just pay attention to the first pair and let the rest follow suit.

4.4.1 Length

For an associative array `aa`, the property `aa.length` of type `size_t` yields the number of keys in `aa` (and also the number of values, given that there is a one-to-one mapping of keys to values). The type of `aa.length` is `size_t`.

A default-constructed associative array has `length` equal to zero and also compares equal to `null`.

```
string[int] aa;
assert(aa == null);
assert(aa.length == 0);
aa = [0:"zero", 1:"not zero"];
assert(aa.length == 2);
```

Unlike the homonym property for arrays, associative arrays' `.length` is not writable. You may, however, write `null` to an associative array to obliterate all of its content.

4.4.2 Reading and Writing Slots

To write a new key/value pair into `aa`, or to overwrite the value currently stored for that key, just assign to `aa[key]` like this:

```

5 // Create a string-to-string associative array
  auto aa = [ "hello":"salve", "world":"mundi" ];
  // Overwrite values
  aa["hello"] = "ciao";
  aa["world"] = "mondo";
10 // Create some new key/value pairs
  aa["cabbage"] = "cavolo";
  aa["mozzarella"] = "mozzarella";

```

To read a value off an associative array given a key, just read `aa[key]`. (The compiler distinguishes reads from writes and invokes slightly different functions.) Continuing the example above:

```

assert(aa["hello"] == "ciao");

```

If you try to read the value for a key not found in the associative array, a range violation exception is thrown. Oftentimes, throwing an exception in case a key doesn't exist is a bit too harsh to be useful, so associative arrays offer a read with default in the form of a two-argument `get` method. In the call `aa.get(key, defaultValue)`, if `key` is found in the map, its corresponding value is returned and `defaultValue` is not evaluated; otherwise, `defaultValue` is evaluated and returned as the result of `get`.

```

assert(aa["hello"] == "ciao");
25 // Key "hello" exist, therefore ignore the second argument
  assert(aa.get("hello", "salute") == "hello");
  // Key "yo" doesn't exist, return the second argument
  assert(aa.get("yo", "buongiorno") == "buongiorno");

```

If you want to peacefully test for the existence of a key in an associative array, use the `in` operator:

```

assert("hello" in aa);
assert("yowza" !in aa);
// Trying to read aa["yowza"] would throw

```

4.4.3 Copying

35 Associative arrays are sheer references with shallow copying: copying or assigning associative arrays just creates new aliases for the same underlying slots. For example:

```

auto a1 = [ "Jane":10.0, "Jack":20, "Bob":15 ];
auto a2 = a1;           // a1 and a2 refer to the same data
a1["Bob"] = 100;       // Changing a1...
5 assert(a2["Bob"] == 100); // ...is same as changing a2...
a2["Sam"] = 3.5;       // ...and vice
assert(a2["Sam"] == 3.5); // versa

```

To create a duplicate of an associative array, use `.dup`, which works the same as for arrays.

```

auto a1 = [ "Jane":10.0, "Jack":20, "Bob":15 ];
auto a2 = a1.dup;      // a1 and a2 are independent
a1["Bob"] = 100;
15 assert(a2["Bob"] == 15); // Changing a1 does not change a2

```

Assigning `null` to an associative array makes it effectively devoid of any information.

4.4.4 Comparing for Equality

The operators `'is'` and `'=='` work the expected way. For two associative arrays of the same type `a` and `b`, the expression `a is b` yields `true` if and only if `a` and `b` refer to the same associative array (e.g., one was initialized as a copy of the other). The expression `a == b` compares the key/value pairs of two arrays with `'=='` in turn. For `a` and `b` to be equal, they must have equal key sets and equal values associated with each key.

```

25 auto a1 = [ "Jane":10.0, "Jack":20, "Bob":15 ];

```

4.4.5 Removing Elements

To remove a key/value pair from the map, pass the key to the `remove` method of the associative array.

```

30 aa.remove("hello");
assert("hello" !in aa);
aa.remove("yowza"); // has no effect: "yowza" was not in aa

```

The `remove` method returns a `bool` that is `true` if the deleted key was in the associative array, or `false` otherwise.

[Note: normally the code below would not be included. It is visible because it fails to corroborate statements made in the text.]

```

unittest {
  int[int] a = [ 1:2 ];

```

```
assert(a.remove(1));
assert(!a.remove(1));
}
```

5 4.4.6 Iterating

You can iterate an associative array by using the good old `foreach` statement (§ 3.8.5 on page 82). The key/value slots are iterated in an unspecified order.

```
import std.stdio;

10 void main() {
    auto coffeePrices = [
        "french vanilla" : 8.75,
        "java" : 7.99,
15     "french roast" : 7.49
    ];
    foreach (kind, price; coffeePrices) {
        writeln("%s costs $%s per pound", kind, price);
    }
20 }
```

The program above will print:

```
french vanilla costs $8.75 per pound
french roast costs $7.49 per pound
25 java costs $7.99 per pound
```

To fetch a copy of all keys in an array, use the `.keys` property. For an associative array `aa` of type `V[K]`, the type returned by `aa.keys` is `K[]`.

```
auto gammaFunc = [-1.5:2.363, -0.5:-3.545, 0.5:1.772];
30 double[] keys = gammaFunc.keys;
assert(keys == [ 0.5, -0.5, -1.5 ]);
```

Similarly, for `aa` of type `V[K]`, the `aa.values` property yields the values stored in `aa` as an array of type `V[]`. Generally, it is preferable to iterate with `foreach` instead of fetching keys of values because the properties allocate a new array, which may have a considerable size for large associative arrays.

4.4.7 User-Defined Types as Keys

Internally, associative arrays use hashing and sorting for keys to ensure fast retrieval of values given keys. For a user-defined type to be used as a key in an associative array, it

must define two special methods, `opHash` and `opCmp`. We haven't yet learned how to define user-defined types and methods, so for now let's defer that discussion to Chapter ??.

4.5 Strings

Strings receive special treatment in D. Two decisions made early in the definition of the language turned out to be winning bets. First, D embraces Unicode as its standard character set. Unicode is today's most popular and comprehensive standard for defining and representing textual data. Second, D chose UTF-8, UTF-16, and UTF-32 as its native encodings, without favoring either and without preventing user code from using other encodings.

In order to understand how D deals with text, we need to acquire some knowledge of Unicode and UTF. For an in-depth treatment, "Unicode Explained" [14] is a useful resource; the Unicode Consortium Standard document, currently at the fifth edition corresponding to version 5.1 of the Unicode standard [31], is the ultimate reference.

4.5.1 Code Points

One important fact about Unicode that, once understood, dissipates a lot of potential confusion, is that Unicode separates the notion of abstract character, or *code point*, from the notion of representation, or *encoding*. This is a non-trivial distinction that often escapes the unwary, particularly because the well-known ASCII standard does not make it. Good old ASCII maps each character commonly used in English text, plus a few "control codes," to a number between 0 and 127—i.e., 7 bits. Since at the time ASCII got introduced most computers already used the 8-bit byte (octet) as a unit of addressing, there was no question about "encoding" ASCII text at all: use 7 bits off an octet, that was the encoding. (The remaining bit left the door open for creative uses, which led to a Cambrian explosion of mutually incompatible extensions.)

Unicode, in contrast, first defines code points, which are, simply put, numbers assigned to abstract characters. The abstract character 'A' receives number 65, the abstract character '€' receives number 8364, and so on. Deciding which symbols deserve a place in the Unicode mapping and how to assign numbers to them is one important task of the Unicode Consortium, and that's great because the rest of us can use the mapping without worrying about the minutiae of defining and documenting it.

As of version 5.1, Unicode code points lie between 0 and 1,114,111 (the upper limit is more often expressed in hexadecimal: `0x10FFFF`, or in Unicode's specific spelling, `U+10FFFF`). A common misconception about Unicode is that two bytes are enough to represent any Unicode character, perhaps because some languages standardized on two-byte characters originating in earlier versions of the Unicode standard. In fact, there are exactly 17 times more Unicode symbols than the 65,536 afforded by a two-byte representation. (Truth be told, most of the higher code points are seldom used or not yet allocated.)

Anyhow, when discussing code points, representation should not necessarily come to mind. At the highest level, code points are a giant tabulated function mapping integers from 0 to 1,114,111 to abstract character entities. There are many details on how that numeric range is allocated, but that does not diminish the correctness of our highest-level description. Exactly how to put Unicode code points in sequences of bytes is something that *encodings* need to worry about.

4.5.2 Encodings

If Unicode simply followed ASCII's grand tradition, it would have just rounded the upper limit $0 \times 10\text{FFFF}$ to the next byte, obtaining a simple three-bytes representation for each code point. This potential representation has an issue, however. Most text in English or other Latin-derived writing systems would use a statistically very narrow range of code points (numbers), which leads to wasted space. The storage for the typical English text would just blow up in size by a factor of three. Richer alphabets such as Asian writing systems would make better use of the three bytes, but that's fine because there would be fewer total symbols in the text (each symbol is more informative).

To address the issue of wasting space, Unicode adopted several *variable-length* encoding schemes. Such schemes use one or more narrow codes to represent the full range of Unicode code points. The narrow codes (usually 8 or 16 bit) are known as *code units*. Each code point is represented by one or more code units.

UTF-8 is the first encoding to be standardized. UTF-8, invented by Ken Thompson in one evening inside a New Jersey diner [23], is an almost canonical example of solid, ingenious design. The basic idea behind UTF-8 is to use 1–6 bytes for encoding any given character, and to add control bits to disambiguate between encodings of different lengths. UTF-8 is identical to ASCII for the first 127 code points. That instantly makes any ASCII text also valid UTF-8 text, which in and by itself was a brilliant move. For code points beyond the ASCII range, UTF-8 uses a variable length encoding shown in Table 4.2.

Table 4.2: UTF-8 encodings. The choice of control bits allows midstream synchronization, error recovery, and backward iteration.

Code point (hex)	Codes (binary)
00000000–0000007F	0xxxxxxx
00000080–000007FF	110xxxxx 10xxxxxx
00000800–0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
00010000–001FFFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
00200000–03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
04000000–7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

Since today the range of defined Unicode code points stops at 0x10FFFF, the last two sequences are reserved for future use; only up to four-byte encodings are currently valid.

The control bit patterns chosen have two interesting properties:

1. A non-leading byte is never equal to a leading byte; and
- 5 2. The first byte unambiguously determines the length of an encoding.

The first property is crucial because it enables two important applications. One is simple synchronization—if you pick up a UTF-8 transmission somewhere in midstream, you can easily figure out where the next code point starts: just look for the next byte with anything but 10 as its most significant bits. The other application is backward iteration—it is easy to go backward in a UTF-8 string without ever getting confused. Backward iteration opens UTF-8 strings to a host of algorithms (e.g., finding the last occurrence of a string in another can be implemented efficiently). The second property is not essential, but simplifies and accelerates string processing.

Ideally, code points with larger absolute values are statistically used less frequently. If that condition is fulfilled, UTF-8 acts as a good statistical encoder by encoding more frequent symbols in less bits. This is certainly the case for Latin-derived languages, where most code units fit in one byte and the occasional accented characters fit in two.

UTF-16 is also a variable-length encoding, but uses a different (and arguably less elegant) approach to encoding. Code points in between 0 and 0xFFFF are encoded as a sole 16-bit code unit, and code points between 0x10000 and 0x10FFFF are represented by a pair in which the first code unit is in the range 0xD800 and 0xDBFF, and the second code unit is in the range 0xDC00 through 0xDFFF. To support this encoding, Unicode allocates no valid characters to numbers in the range 0xD800 through 0xDBFF. The two ranges are called high surrogate area and low surrogate area, respectively.

One criticism commonly leveled against UTF-16 is that it makes the statistically rare cases also the most complicated and the ones deserving the most scrutiny. Most—but alas, not all—Unicode characters (the so-called Basic Multilingual Plane) *do* fit in one UTF-16 code unit, and therefore a lot of UTF-16 code tacitly assumes one code unit per character and is effectively untested for surrogate pairs. To further the confusion, some languages initially centered their string support around UCS-2, a precursor of UTF-16 with exactly 16 bits per code point, to later add UTF-16 support, subtly obsoleting older code that relied on a one-to-one mapping between characters and codes.

Finally, UTF-32 uses 32 bits per code unit, which allows a true one-to-one mapping of code points to code units. This means UTF-32 is the simplest and easiest to use representation, but it's also the most space-consuming. A common recommendation is to use UTF-8 for storage and UTF-32 temporarily during processing if necessary.

4.5.3 UTF Character Types

D defines three character types: `char`, `wchar`, and `dchar`, representing code units for

UTF-8, UTF-16, and UTF-32 respectively. Their `.init` values are intently invalid encodings: `char.init` is `0xFF`, `wchar.init` is `0xFFFF`, and `dchar.init` is `0x0000FFFF`. Table 4.2 on page 127 clarifies that `0xFF` may not be part of any valid UTF-8 encoding, and also Unicode deliberately assigns no valid code point for `0xFFFF`.

5 Used individually, the three character types are mostly behaving like unsigned integers and can occasionally be used to store invalid UTF code points (the compiler does not enforce valid encodings throughout), but the intended meaning of `char`, `wchar`, and `dchar` is as UTF code points. For general 8-, 16-, and 32-bit unsigned integers, or for using other encodings than UTF, it's best to use `ubyte`, `ushort`, and `uint` respectively. For
 10 example, if you want to use pre-Unicode 8-bit code pages, you may want to use `ubyte`, not `char`, as your building block.

4.5.4 Arrays of Characters + Benefits = Strings

When assembling any of the character types in an array—as in `char[]`, `wchar[]`, or `dchar[]`—the compiler and the runtime support library “understand” that you are
 15 working with UTF-encoded Unicode strings. Consequently, arrays of characters enjoy the power and versatility of general arrays, plus a few extra goodies as Unicode denizens.

In fact, D already defines three string types corresponding to the three character widths: `string`, `wstring`, and `dstring`. They are not special types at all, in fact they are aliases for character array types, with a twist: the character type is adorned with the
 20 `immutable` qualifier to disallow arbitrary change of individual characters in strings. For example, type `string` is a synonym for the more verbose type `immutable(char)[]`. We won't get to discussing type qualifiers such as `immutable` until Chapter ??, but for strings of all widths the effect of `immutable` is very simple: a `string` a.k.a. a `immutable(char)[]` is just like a `char[]` (and a `wstring` is just like a `wchar[]` etc.), except you can't assign new
 25 values to individual characters in the string:

```
string a = "hello";
char h = a[0];           // fine
a[0] = 'H';             // Error!
                        // Cannot assign to immutable(char)!
```

30 To change one individual character in a string, you need to create another string via concatenation:

```
string a = "hello";
35 a = 'H' ~ a[1 .. $]; // Fine, makes a == "Hello"
```

Why such a decision? After all, in the case above it's quite a waste to allocate a whole new string (recall from § 4.1.6 on page 108 that `~` always allocates a new array) instead of just modifying the existing one. There are, however, a few good reasons for disallowing modification of individual characters in strings. One reason is that `immutable`

simplifies situations when `string`, `wstring`, and `dstring` objects are copied and then changed. Effectively `immutable` ensures no undue aliasing between strings. Consider:

```

string a = "hello";
5 string b = a;           // b is also "hello"
string c = b[0 .. 4]; // c is "hell"
// If this were allowed, it would change a, b, and c
// a[0] = 'H';
// The concatenation below leaves b and c unmodified
10 a = 'H' ~ a[1 .. $];
assert(a == "Hello" && b == "hello" && c == "hell");

```

With `immutable` characters, you know you can have several variables refer to the same string, without fearing that modifying one would also modify the others. Copying `string` objects is very cheap because it doesn't need to do any special copy management (such as eager copy or copy-on-write).

An equally strong reason for disallowing code unit-level changes in strings is that such changes don't make much sense anyway. Elements of a `string` are variable-length, and most of the time you want to replace logical characters (code points), not physical `chars` (code units), so you seldom want to do surgery on individual `chars`. It's much easier to write correct UTF code if you forgo individual `char` assignments and you focus instead of manipulating entire strings and fragments thereof. However, UTF code is not trivially easy to write; for example, the concatenation `'H' ~ a[1 .. $]` above has a bug in the general case because it assumes that the first code point in `a` has exactly one byte.

The correct way to go about it is

```
a = 'H' ~ a[stride(a, 0) .. $];
```

The function `stride`, found in the standard library module `std.utf`, returns the length of the code starting at a specified position in a string. (To use `stride` and related library artifacts, insert the line

```
30 import std.utf;
```

near the top of your program.) In our case, the call `stride(a, 0)` returns the length of the encoding for the first character (a.k.a. code point) in `a`, which we pass to select the offset marking the beginning of the second character.

A very visible artifact of language's support for Unicode can be found in string literals, which we've already looked at (§ 2.2.5 on page 46). `D` string literals understand Unicode code points and automatically encode them appropriately for whichever encoding scheme you choose. For example:

```
40 import std.stdio;

void main() {
```

```

string a = "No matter how you put it, a \u03bb costs \u20AC20.";
wstring b = "No matter how you put it, a \u03bb costs \u20AC20.";
dstring c = "No matter how you put it, a \u03bb costs \u20AC20.";
writeln(a, '\n', b, '\n', c);
5 }

```

Although the internal representations of `a`, `b`, and `c` are very different, you don't need to worry about that because you express the literal in an abstract way by using code points. The compiler takes care of all encoding details, such that in the end the program
10 prints three lines containing the same exact text:

```
No matter how you put it, a λ costs €20.
```

The encoding of the literal is determined by the context in which the literal occurs. In the cases above, the compiler has the literal morph without any run-time processing into the encodings UTF-8, UTF-16, and UTF-32 (corresponding to types `string`,
15 `wstring`, and `dstring`), in spite of it being spelled the exact same way throughout. If the requested literal encoding is ambiguous, suffixing the literal with one of `'c'`, `'w'`, or `'d'` (something "like that" `d`) forces the encoding of the string to UTF-8, UTF-16, and UTF-32 respectively (refer to § 2.2.5.2 on page 48).

4.5.4.1 String Concatenation

20 Since all string widths ultimately represent the same information—sequences of code points—the language relaxes the typing rules pertaining to the concatenation operators `'~'` and `'~='`. You may concatenate a string of any width with another string of any other width, and also with a character of any width.

[Note: the code below does not work due to a bug in the current compiler.]

```

25 string a = "Hall\u00E5";
wstring b = ", ";
dstring c = "v\u00E4rd";
auto d = b ~ c;           // d has type wstring, same as b
30 a ~= d ~ '!';         // concatenate string with character
writeln(a);

```

The program above prints "Hello, world!" in Swedish:

```
Hallå, värd!
```

35 Concatenating strings of different widths with `lhs ~ rhs` returns a string of the same width as `lhs`. If `lhs` is a character, the result of concatenation has the same type as `rhs`. It is illegal to concatenate two characters.

The append operator `'~='` enjoys similar privileges. For a string value `a` and a string or character value `b`, `a ~= b` is equivalent to `a = a ~ b`, just more efficient because it may

benefit from in-place expansion. It may appear odd at first, but ‘~=' works for strings containing `immutable` code points (i.e., `string`, `wstring`, and `dstring`): appending to a string does not change what is already there.

4.5.4.2 foreach with Strings

5 If you iterate a string `str` of any width like this:

```
foreach (c; str) {
    ... // use c
}
```

10 then `c` will iterate every code *unit* of `str`. For example, if `str` is an array of `char` (be they `immutable` or not), `c` takes type `char`. This is expected from the general behavior of `foreach` with arrays, but is sometimes undesirable for strings. For example, let's print each character of a string enclosed in square brackets:

```
void main() {
    string str = "Hall\u00E5, v\u00E4rd!";
    15 foreach (c; str) {
        write('[', c, ']');
    }
    writeln();
}
```

20 The program above ungainly prints:

```
[H][a][l][l][?][?][,][ ][v][?][?][r][d][!]
```

25 The reverse video ‘?’ (may vary depending on system and font used) is the console's mute way of protesting against seeing an invalid UTF code. Of course—trying to print alone a `char` that would make sense only in combination with another `char` is bound to fail.

The interesting part starts when you specify a different character type for `c`. For example, specify `dchar` for `c`:

```
... as above, just add "dchar" ...
30 foreach (dchar c; str) {
    write('[', c, ']');
}
```

In this case, the compiler inserts automatically code for transcoding on-the-fly each code unit in `str` in the representation dictated by `c`'s type. The loop above prints:

```
[H][a][l][l][å][,][ ][v][ä][r][d][!]
```

which indicates that the double-byte characters 'á' and 'ä' were converted correctly to one `dchar` each, and subsequently printed correctly. The same exact result would be printed if `c` had type `wchar` because the two non-ASCII characters used fit in one UTF-16 unit each, but not in the most general case (surrogate pairs would be wrongly processed). To be on the safe side, it is of course best to use `dchar` with loops over strings.

In the case above, the transcoding performed by `foreach` went from a narrow to a wide representation, but it could go either way. For example, you could start with a `dstring` and iterate it one (encoded) `char` at a time.

4.6 Summary

- ¹⁰ Table 9.1 on page 301 summarizes dynamic array operations; Table 4.4 on page 135 summarizes operations on fixed-size arrays; and Table 4.5 on page 136 summarizes operations available for associative arrays.

Table 4.3: Dynamic array operations (a and b are two values of type T[], t, t₁, ..., t_k are values of type T, and n is a value convertible to type size_t).

Name	Type	Description
<code>new T[n]</code>	T[]	Creates array (§ 4.1 on page 101)
<code>[t₁, t₂, ..., t_k]</code>	T[]	Array literal, T is deduced as the type of t ₁ (§ 2.2.6 on page 50, § 4.1 on page 101)
<code>a = b</code>	T[]	Assigns array to array (§ 4.1.4 on page 106)
<code>a[⟨e⟩]</code>	ref T	Accesses element by index ('\$' is replaced in ⟨e⟩ with a.length, ⟨e⟩ must be convertible to size_t, and ⟨e⟩ < a.length) (§ 4.1 on page 101)
<code>a[⟨e₁⟩ .. ⟨e₂⟩]</code>	T[]	Takes a slice off a ('\$' is replaced in ⟨e ₁ ⟩ and ⟨e ₂ ⟩ with a.length, ⟨e ₁ ⟩ and ⟨e ₂ ⟩ must be convertible to size_t, and ⟨e ₁ ⟩ <= ⟨e ₂ ⟩ && ⟨e ₂ ⟩ <= a.length) (§ 4.1.3 on page 105)
<code>a[]</code>	T[]	Participate in array-wise expressions (§ 4.1.7 on page 108), otherwise just the identity operation a[0 .. \$]
<code>a.dup</code>	T[]	Duplicates the array (§ 4.1 on page 101)
<code>a.length</code>	size_t	Reads array's length (§ 4.1.10 on page 114)
<code>a.length = n</code>	size_t	Changes array's length (§ 4.1.1 on page 103)
<code>a is b</code>	bool	Compares arrays for identity (§ 4.1.5 on page 107, § 2.4.14.3 on page 64)
<code>a !is b</code>	bool	Same as !(a is b)
<code>a == b</code>	bool	Compares arrays for element-for-element equality (§ 4.1.5 on page 107, § 2.4.7 on page 56)
<code>a != b</code>	bool	Same as !(a == b)
<code>a ~ t</code>	T[]	Concatenates array with one value
<code>t ~ a</code>	T[]	Concatenates one value with array
<code>a ~ b</code>	T[]	Concatenates two arrays
<code>a ~= t</code>	T[]	Appends element to array
<code>a ~= b</code>	T[]	Appends array to array
<code>a.ptr</code>	T*	The address of a's first element (unsafe)

Table 4.4: Fixed-size array operations (a and b are two values of type T[], t, t₁, ..., t_k are values of type T, and n is a statically-known value convertible to type size_t).

Name	Type	Description
[t ₁ , ..., t _k]	T[k]	Array literal, but only if type T[k] is explicitly requested. T is deduced as the type of t ₁ (§ 2.2.6 on page 50, § 4.1 on page 101)
a = b	ref T[n]	Copies contents over (§ 4.2.4 on page 117)
a[⟨e⟩]	ref T	Accesses element by index ('\$' is replaced in ⟨e⟩ with a.length, ⟨e⟩ must be convertible to size_t, and ⟨e⟩ < a.length) (§ 4.1 on page 101)
a[⟨e ₁ ⟩ .. ⟨e ₂ ⟩]	T[]	Takes a slice off a ('\$' is replaced in ⟨e ₁ ⟩ and ⟨e ₂ ⟩ with a.length, ⟨e ₁ ⟩ and ⟨e ₂ ⟩ must be convertible to size_t, and ⟨e ₁ ⟩ <= ⟨e ₂ ⟩ && ⟨e ₂ ⟩ <= a.length) (§ ?? on page ??)
a[]	T[]	Participate in array-wise expressions (§ 4.1.7 on page 108) or just convert a to dynamic array, same as a[0 .. \$]
a.dup	T[]	Duplicates the array (§ 4.2.4 on page 117)
a.length	size_t	Reads array's length (§ 4.2.1 on page 116)
a is b	bool	Compares arrays for identity (§ 4.2.5 on page 118, § 2.4.14.3 on page 64)
a !is b	bool	Same as !(a is b)
a == b	bool	Compares arrays for element-for-element equality (§ 4.2.5 on page 118, § 2.4.7 on page 56)
a != b	bool	Same as !(a == b)
a ~ t	T[]	Concatenates array with one value
t ~ a	T[]	Concatenates one value with array
a ~ b	T[]	Concatenates two arrays
a.ptr	T*	The address of a's first element (unsafe)

Table 4.5: Associative array operations (a and b are two values of type $V[K]$, k, k_1, \dots, k_k are values of type K , and v, v_1, \dots, v_k are values of type V).

Name	Type	Description
$[t_1:v_1, \dots, t_k:v_k]$	$V[K]$	Associative array literal; K is deduced as the type of k_1 and V is deduced as the type of v_1 (§ 2.2.6 on page 50, § 4.4 on page 121)
$a = b$	$V[K]$	Assigns b to a (§ 4.4.3 on page 123)
$a[k]$	V	Accesses element by index (if k not found, throws an exception) (§ 4.4.2 on page 123)
$a[k] = v$	V	Associates value v with key k (overwrites the previous association if any) (§ 4.4.2 on page 123)
$k \text{ in } a$	V^*	Looks up k in a , returns <code>null</code> if absent or a pointer to the value associated to k if present (§ 4.4.2 on page 123)
$k \text{ !in } a$	<code>bool</code>	Same as $!(k \text{ in } a)$
$a.\text{dup}$	$V[K]$	Duplicates the associative array (§ 4.4.3 on page 123)
$a.\text{length}$	<code>size_t</code>	Reads the number of elements in a (§ 4.4.1 on page 122)
$a \text{ is } b$	<code>bool</code>	Compares associative arrays for identity (§ 4.4.4 on page 124, § 2.4.14.3 on page 64)
$a \text{ !is } b$	<code>bool</code>	Same as $!(a \text{ is } b)$
$a == b$	<code>bool</code>	Compares arrays for element-for-element equality (§ 4.4.4 on page 124, § 2.4.7 on page 56)
$a != b$	<code>bool</code>	Same as $!(a == b)$
$a.\text{remove}(k)$	<code>bool</code>	Remove the association of k , if any; returns <code>true</code> if and only if k existed in a
$a.\text{get}(k, v)$	V	Returns the value in a corresponding to key k