V language types Cheat Sheet

by Durobot via cheatography.com/148384/cs/32362/

Primitive types

bool	8 bits, boolean
string	An **immutable** array of bytes. All Unicode characters are UTF-8. Indexing a string produces a byte, not a rune nor another string. Indexes correspond to bytes in the string, not Unicode code points. Use single or double quotes to denote strings.
rune	32 bits, unicode code point (~character), alias for u32. To denote runes, use ` (backticks).
i8	8 bits, signed integer
i16	16 bits, signed integer
int	32 bits, signed integer (the default int type)
i64	64 bits, signed integer
u8	8 bits, unsigned integer (alias for byte?)
u16	16 bits, unsigned integer
u32	32 bits, unsigned integer
u64	64 bits, signed integer
isize	System-dependent signed integer, 64 bits on x64
usize	System-dependent unsigned integer, 64 bits on x64
f32	32 bits, floating-point number
f64	64 bits, floating-point number (the default float type)

Runes

Can be converted to a UTF-8 string by using the **.str()** method:

```
rocket := `🚀 `
assert rocket.str() == '🚀'
```

Can be converted to UTF-8 bytes by using the **.bytes()** method:

```
rkt := `🚀`
assert rkt.by tes() == [u8(0x f0) ,0x 9f, 0x9 a,0x8 0]
```

Hex, Octal, and Unicode escape sequences work in rune literals:

```
assert `\x61` == `a`
assert `\141` == `a`
assert `\u0061` == `a`
```

Multibyte literals work too:

```
assert `\u2605` == `★`
assert `\u260 5`.b ytes() == [u8(0xe2), 0x98, 0x85]
assert `\xe2 \x98 `.b ytes() == [u8(0xe2),0x98]
assert `\342 \230 `.b ytes() == [u8(0x e2) ,0x98]
```

String interpolation

String interpolation (cont)

Format specifier pattern

```
${varn ame :[f lag s][ wid th] [.p rec isi on] [type]
```

* flags: **-** to left-align output within the field, **+** to right-align (default), **0** (z
as the padding character instead of the default space
* width: integer value - minimum width of total field to output
* precision: an integer value preceded by a . Guarantees the number of
the decimal point, if the input is a float. Ignored if integer.
* type: **f** or **F** - float, rendered as float; **e** or **E** - float, rendered as expone
foat, small values rendered as float, large values as exponent; **d** - integ
as decimal; **x** or **X** - integer, rendered as hexadecimal; **o** - integer, rend
b - integer, rendered as binary; **s** - string (almost never used).
Format strings are parsed at compile time, specifying type can help det

```
x := 123.4567
'[${int(x):010}]' // left-pad with 0 => [0000000123]
'[${10.0000:.2}]' // strip insign ificant 0s at the e
]
 printl n(' ${1 0.1 234 :.2}') // => 10.12
println('[${10.0000:.2f}]') // show 0s at the end
```

Arrays

A collection of data elements of the same type:

```
mut nums := [1, 2, 3]
println(nums[0]) // `1` - zero-indexed
nums << 4 // appending element, [1, 2, 3, 4]
nums << [5, 6] // appending array, [1, .., 5, 6]
1 in nums // true
x := nums[999] // panic, use or { default value }
```

Arrays: Fields (read-only)

len: length - the number of pre-allocated and initialized elements in the array;

cap: capacity - the amount of memory space which has been reserved for elements, but not initialized or counted as elements. The array can grow up to this size without being reallocated.

```
mut nums := [1, 2, 3]
println(nums.len) // "3"
println(nums.cap) // " 3" or greater
nums = [] // The array is now empty
println(nums.len) // " 0"
```

data is a field (of type voidptr) with the address of the first element. This is for low-level unsafe code.

Arrays: Initialization

Use **$** before a variable name. The variable will be converted to a string and embedded into the literal:

```
name := 'Bob'
println('Hello, $name!') // Hello, Bob!
```

Works with fields: `'age = $user.age'`

And with expressions: `'can register = ${user.age > 13}'`

Array type is determined by the first element:

`[1, 2, 3]` is an array of ints (`[]int`)

`['a', 'b']` is an array of strings (`[]string`)

`[u8(16), 32, 64]` is an array of u8 (`[]u8`)

Second initialization syntax:

```
mut a := []int{len: 10000, cap: 30000, init: 3}
```

len (default 0), **cap** and **init** (type default) are optional

```
users := []int{} // empty array
c := []int{len: 3, init: it} // [0, 1, 2] - it=index
```

By **Durobot**
cheatography.com/durobot/

Not published yet.
Last updated 29th June, 2022.

V language types Cheat Sheet

by Durobot via cheatography.com/148384/cs/32362/

Arrays: Multidimensional

```
mut a := [][]in t{len: 2, init: []int{len: 3}}
a[0][1] = 2
mut a := [][][] int {len: 2, init: [][]in t{len: 3, init: [][]int{len: 4}}
}}}
a[0][1][1] = 2
```

Fixed size arrays

Arrays with fixed size (length). You cannot append elements to them, nor shrink them. You can only modify their elements in place. Access to the elements of fixed size arrays is more efficient, they need less memory than ordinary arrays, and unlike ordinary arrays, their data is on the stack, so you may want to use them as buffers if you do not want additional heap allocations.

Most methods are defined to work on ordinary arrays, not on fixed size arrays. You can convert a fixed size array to an ordinary array with slicing:

```
mut fnums := [3]int{} // fnums is a fixed size array with 3 elements.
fnums[0] = 1
fnums[1] = 10
fnums[2] = 100
println(fnums) // => [1, 10, 100]
println(typeof(fnums).name) // => [3]int
fnums2 := [1, 10, 100]! // short init syntax that does the same (the syntax will probably change)
anums := fnums[..] // same as `anums := fnums[0..fnums.len]`
println(anums) // => [1, 10, 100]
println(typeof(anums).name) // => []int
```

Note that slicing will cause the **data** of the fixed size array to be **copied to the newly created ordinary array**.

Structs

Structs: Trailing struct literal arguments

```
my_fn( text: 'Click me', width: 100) // same as
my_fn(BtnCfg{text: 'Click me', width: 100})
```

Note if struct `BtnCfg` is annotated with `[params]`, `my_fn` can be called without explicit parameters, as `my_fn()` (default field values are used).

Structs: Methods

```
println(usr.can_register()) // true
```

Structs: Heap structs

Use **&** to alloc on the heap and get reference

```
p := &P oin t{10, 10} // The type of p is &Point
println(p.x) // Same syntax for field access
```

```
struct Foo {
mut:
    x int
}
fa := Foo{1} // On stack
mut a := fa // a is a copy of fa
a.x = 2
assert fa.x == 1
assert a.x == 2
```

```
mut fc := Foo{1} // Must be mut, as we
mut c := &fc // can't have mutable ref to immutable
c.x = 2
assert fc.x == 2
println(c) // &Foo{ x: 2 }, Note `&`
```

Structs: Embedded

Structs are allocated on the stack.

```
struct Point {
  x int // immutable
  y int // immutable
}
mut p := Point {
  x: 10
  y: 20
}
println(p.x) // Struct fields access
p = Point{10, 20} // Alt syntax when <= 3 fields
```

Struct fields are private (accessible within the module) and immutable by default. This can be changed with **pub** and **mut** modifiers. There are 5 options in total:

```
struct Foo {
  a int // private immutable (default)
mut:
  b int // private mutable
  c int // another one
pub:
  d int // public immutable (readonly)
pub mut:
  e int // public, mutable only in parent module
__global: // Not recommended
  f int // public and mutable everywhere
}
```

Structs: Default field falues

```
struct Foo { // fields are zeroed by default
  n   int    // n is 0 by default
  s   string // s is '' by default
  a   []int  // a is allocated (`[]in t{}`) by defaul
t
  p int = -1 // custom default value
  r   int [required] // must be initia lized by hand
}
_ = Foo{} // Error, r must be initia lized explicitly
```

```
struct Size {
mut:
  width  int
  height int
}
fn (s &Size) area() int { return s.width * s.height }
struct Button {
  Size // Size fields /me thods embedded in Button
  title string
}
mut button := Button {
  title: 'Click me'
  height: 2
}
button.width = 3
assert button.area() == 6 // Use explicit struct name
assert button.Si ze.a rea() == 6 // ambiguous fields
```

Initialize embedded struct:

```
mut button := Button {
  Size: Size {
    width: 3
    height: 2
  }
}
```

Or assign values:

```
button.Size = Size {
  width: 4
  height: 5
}
```

Numbers

```
a := 123 // Declare a new int variable, assigning 12
```

```
a := 0x7B // Same int variable in hexade cimal notat
n
b := 0b01111011 // Same in binary notation
c := 0o173 // Same in octal notation
```

_ can be used as a separator: `1_000_000 0b0_11 3_122.55 0xF_ 0o17_3`

Numbers (cont)

To declare variables of types other than int, use casting:

```
a := i64(123)
b := u8(42)
c := i16(12345)
f := 1.0 // f64 is the default float type
f1 := f64(3.14) // f64 is the default float type
f2 := f32(3.14)
f3 := 42e1 // 420
f4 := 123e-2 // 1.23
f5 := 456e+2 // 45600
```

Strings

```
name := 'Bob'
assert name.len == 3 // number of bytes
name[0] == u8(66) // indexing gives a byte
name[1..3] == 'ob'  // slicing -> string 'ob'
cr_lf := '\r\n' // escape special chars like in C
assert '\xc0'[0] == u8(0xc0) // `\x##` hex number
aardvark := '\141a rdvark' // \### octal
star_str := '\u2605' // ★ Unicode as hex `\u####`
assert star_str == '\xe2 \x98 \x85' // UTF-8
```

For **raw strings**, prepend **r**. Escape handling is skipped:

```
s := r'hell o\n world' // the `\n` is two character
s
```

String operators

```
bobby := 'Bob' + 'by' // "Bobby"
mut s := 'hello '
s += 'world' // " hello world"
println('age = ' + 10.str()) // must be same type
```

Converting strings to integers

```
'42'.int() == 42 // all
'0xc3'.int() == 195 // int
'0o10'.int() == 8 // literals
'0b1111_0000_1010'.int() == 3850 // are
'-0b1111_0000_1010'.int() == -3850 // supported
```

Converting strings to runes

```
'Bob'.r unes() // == [`B`, `o`, `b`]
```

Converting strings to bytes

```
'Bob'.b ytes() // == [u8(66), 111, 98]
```

Array methods

Array methods (cont)

a.map(it.to_upper())	produces a new array, each element is the value of the expression in parenthethes (**it** refers to every element of the original array, consecutively). Can accept anonymous function: `a.map(fn (w string) string { return w.to_u pper() })`
a.repeat(n)	concatenates the array elements n times
a.insert(i, val)	inserts a new element val at index i and shifts all following elements to the right
a.insert(i, [3, 4, 5])	inserts several elements
a.prepend(val)	inserts a value at the beginning, equivalent to a.insert(0, val)
a.prepend(arr)	inserts elements of array arr at the beginning
a.trim(new_len)	truncates the length (if new_length < a.len, otherwise does nothing)
a.clear()	empties the array without changing cap (equivalent to a.trim(0))
a.delete_many(start, size)	removes size consecutive elements from index start – triggers reallocation
a.delete(index)	a.delete_many(index, 1)
a.delete_last()	removes the last element
a.first()	equivalent to a[0]
a.last()	equivalent to a[a.len - 1]
a.pop()	removes the last element and returns it
a.reverse()	makes a new array with the elements of a in reverse order
a.reverse_in_place()	reverses the order of elements in a
a.join(joiner)	concatenates an array of strings into one string using joiner string as a separator

a.str()	converts a to string
a.clone()	produces a new array, containing a copy of data in the original array
a.filter(it % 2 == 0)	produces a new array, containing a copy of elements of the original, that make the expression evaluate to true (**it** refers to every element of the original array, consecutively). Can accept anonymous function: `a.filt er(fn (x int) bool { return x % 2 == 0 })`

Cheatography

V language types Cheat Sheet

by Durobot via cheatography.com/148384/cs/32362/

Array methods (cont)

a.all(it == 2)	true if the expression evaluates to true for all elements of a. (**it** refers to every element of the array, consecutively)
a.any(it > 2)	true if the expression evaluates to true for at least one element of a. (**it** refers to every element of the array, consecutively)

Array methods: Chaining

```
// using filter, map and negatives array slices
files := ['pipp o.jpg', '01.bmp', '_v.txt', 'img_0 2.jpg', 'img_0 1.JPG']
filtrd := files.f il ter (it #[- 4..].t o_l ower() == '.jpg').map(it.to_u
pper())
// ['PIPP O.JPG', 'IMG_0 2.JPG', 'IMG_0 1.JPG']
```

Array slices: Negative indexing

Negative indexing starts from the end of the array towards the start, for example **-3** is equal to **array.len - 3**. Negative slices have a different syntax: **a#[..-3]**. The result is "locked" inside the array.The returned slice is always a valid array, though it may be empty:

```
a := [0, 1, 2, 3, 4, 5]
a#[-3..] // [3, 4, 5]
a#[-20..] // [0, 1, 2, 3, 4, 5]
a#[-20..-3] // [0, 1, 2]
a#[..-1] // [0, 1, 2, 3, 4]
// Empty arrays
a#[-20..-6] // []
a#[20..10] // []
a#[20..30] // []
```

Array slices

Maps

Keys can be **string**s, **rune**s, **integer**s, **float**s or **voidptr**s.

```
mut m := map[st rin g]int{} // string keys and int value
m['one'] = 1
m['two'] = 2
println(m['one']) // "1"
println(m['bad_key']) // "0"
println('bad_key' in m) // `in` to detect if such key
println(m.keys()) // ['one', 'two']
m.delete('two')
```

Map can be initialized using this short syntax:

```
numbers := { 'one': 1  'two': 2 }
```

If a key is not found, a zero value is returned by default:

```
println({ 'abc': 'xyz' }['bad key']) // ''
println({ 1: 123, 2: 456 }[3]) // 0
```

Or use an **or {}** block to handle missing keys:

```
mm := map[string]int{}
val := mm['ba d_key'] or { panic('key not found') }
```

Check if a key is present and get its value (if present):

```
m := { 'abc': 'def' }
if v := m['abc'] {  printl n('the map value for that
```

Unions

A slice is a part of a parent array. Initially it refers to the elements between two indices separated by a .. operator. If a right-side index is absent, it is assumed to be the array length. If a left-side index is absent, it is assumed to be 0.
Slices are arrays themselves (they are not distinct types).

```
nums := [0, 10, 20, 30, 40]
nums[1..4] // [10, 20, 30]
nums[..4] // [0, 10, 20, 30]
nums[1..] // [10, 20, 30, 40]
```

A slice is always created with the smallest possible capacity **cap == len**. Because of this, when the size of the slice increases, it is immediately reallocated and copied to another memory location, becoming independent from the parent array (copy on grow). In particular pushing elements to a slice does not alter the parent:

```
mut a := [0, 1, 2, 3, 4, 5]
mut b := a[2..4]
b[0] = 7 // `b[0]` is referring to `a[2]`
println(a) // [0, 1, 7, 3, 4, 5]
b << 9 // b is reallocated
println(a) // [0, 1, 7, 3, 4, 5] no change
println(b) // [7, 3, 9]
```

Appending to the parent array may or may not make it independent from its child slices. The behaviour depends on the parent's capacity and is predictable:

```
mut a := []int{len: 5, cap: 6, init: 2}
mut b := a[1..4]
a << 3 // no realloc - fits in cap
b[2] = 13 // a[3] is modified
a << 4 // cap exceeded, reallocated
b[1] = 3 // no change in a
println(a) // [2, 2, 2, 13, 2, 3, 4]
println(b) // [2, 3, 13]
```

Call **.clone()** on the slice, if you want an independent copy right away:

```
mut a := [0, 1, 2, 3, 4, 5]
mut b := a[2..4].clone()
b[0] = 7 // NOT referring to a[2]
println(a) // [0, 1, 2, 3, 4, 5]
println(b) // [7, 3]
```

Just like structs, unions support embedding.

```
struct Rgba32 _Co mponent {
  r byte
  b byte
  g byte
  a byte
}
union Rgba32 {
  Rgba32_Component
  value u32
}
clr1 := Rgba32 { value: 0x008811FF }
clr2 := Rgba32 {
  Rgba32_Component: Rgba32 _Co mponent { a: 128 }
}
sz := sizeof(Rgba32)
unsafe { // Union member access must be in unsafe bl
  println('Size: ${sz}B, clr1.b: $clr1.b, clr2.b: $c
}
```

Note that the embedded struct arguments are not necessarily stored in listed.