# Chapter 9 – Structs, Methods and Interfaces

V supports *user-defined* or *custom types* in the form of structs and alias types (see $ 4.2.5)

A struct tries to represent a real-world entity with its properties.

Structs are *composite types,* to use when you want to define a type which consists of a number of properties, each having their own type and value, grouping pieces of data together. Then one can access that data as if it were part of a single entity.

The component pieces of data that constitute the struct type are called *fields.* A field has a type and a name; field names within a struct must be unique.

The concept is called ADT (Abstract Data Type) in some texts on software engineering, it was called a *record* in older languages like Cobol, and it also exists under the same name of *struct* in the C-family of languages, in the OO languages as a lightweight-*class*. In V, structs can also have methods.

However because V does not have the concept of a class, the struct type has a much more important place in V: because there are no globals, data have to be defined in a struct.

## 9.1 Definition of a struct

The general format of the definition of a *struct* is as follows:

**struct** Identifier {

field1 type1

field2 type2

…

}

The fields in this struct have *names*, like field1, field2, and so on. These fields can be of any type, even structs themselves (see § 9.5), functions or interfaces (see Chapter 10). Every field has to be declared separately, even if there are fields with the same type.

Semantically, an array could be seen as a sort of struct but with indexed rather than named fields.

Struct names start with an uppercase letter, and they must have more than one character (otherwise you get a warning). A type Struct1 must be unique in the package mod1 in which it is defined, its complete type name is: mod1.Struct1

Struct fields cannot contain uppercase letters, use snake\_case instead.

You can use a comma to separate fields, but this is optional.

A variable of a struct type T in OO jargon is commonly called an *instance* or *object* of the type T, but in V it is called simply a *value* of type T.

A simple example is given in Listing 10.1 – structs\_fields.v :

struct Struct1 { // (1)

  f1 f32

  str string

mut:

  i1 int

}

fn main() {

// empty struct:

empty := **Struct1{}** // (2)

println(empty.i1) // => 0

println(empty.f1) // => 0.000000

println(empty.str) //

mut ms := **Struct1 {** // (3)

**i1: 7**

**f1: 3.14159**

**str: 'V rules'**

**}**

println(**ms.i1**) // => 7

println(ms.str) // => V rules

println(ms.f1) // => 3.141590

println(ms) // see Output below

// change instance:

ms.i1 = 42

println(ms.i1) // => 42

ms2 := **&Struct1 {i1: 10, f1: 15.5, str: 'Chris'} // (4)**

println(ms2) // 0000000000AF6760

}

Output:

0

0.000000

7

V rules

3.141590

{

  f1: 3.141590

  str: V rules

  i1: 7

}

42

0000000000AF6760

A struct first has to be declared as a type in line (1), then struct values ( *struct-literals*) can be made (as in (2) or (3)), this can also be done on one line.

An empty struct can be defined as empty := Struct1{} : as you can see, its fields are by default initialised to zero values.

The values of the struct-fields can be retrieved with the traditional OO dot-notation: **structname.fieldname**

The fields can also be given a different value with an assignment like this:  **structname.fieldname = value**

Struct fields are by default immutable, if you want to change a field’s value, the field itself must be preceded with the indication mut: see § 9.3.

*Structs have a default str() method*, so they can be printed out with println(). If you want println to behave differently, they you can define your own version of str(), which will override the default.

The struct value ms2 in line (4) given by *&Struct1{ … }* is *allocated on the heap*: ms2 is of type \*Struct1.

By prefixing a struct value with the address-of operator & we get a reference to the struct value. Then the value is allocated on the heap and is automatically cleaned up by V at the end of the function, since it’s scope is this function.

In order to access the fields of a struct, whether the variable is of the struct type or a pointer to the struct type, we use the same notation, as you can see in this example of a 2-dimensional vector or Point:

Listing 10.2 – struct\_point.v:

struct Point {

x f64

y f64

}

const (

origin = Point{x: 0 , y: 0}

)

fn main() {

println(origin)

p1 := Point{x:5, y:10}

p2 := Point{

x: 10

y: 20

}

println(p1.x) // => 5.000000

println(p2.y) // => 20.000000

ptr\_point := &Point{x:10, y:10}

println(ptr\_point.x) // => 10.000000

ptr2\_point := &p2

println(ptr2\_point.y) // => 20.000000

p3 := Point{5, 10}

        println(p3)

}

As you can see, a struct can also be instantiated with a one-line syntax. A struct value can also be a constant, as for origin. Also as you can see with p3, you can leave out the field names. Then the values must be in the order of the fields, and are assigned this way.

The struct fields can also be accessed from the pointer with the exact same dot notation.

Remark: Although there are multiple ways for giving values to the fields of the struct, we’ll use explicitly stated attribute names before values. This allows us to specify the attributes in an unordered way and it’s quite important in the long run from the project maintainability perspective due to no requirement of keeping the order of the [struct](https://vlang.io/docs#structs) attributes (which may change by including a new feature).

A struct can also contain one or more array fields, like for example:

struct Figure {

  corners []Point

}

Here is another example of a 2 dimensional f64 array contained in a struct (see multidim\_struct.v):

struct MultArray {

pub mut:

  data [][]f64

}

fn main () {

    mut mat := MultArray {

        data :  [[f64(11),12,13]

                ,[f64(21),22,23]

                ,[f64(31),32,33]

                ,[f64(41),42,43]]

    }

    mat.data << [[f64(98),97,96,95,94,93]]

    b := mat.data

    c := b[4][4]

    println( c )  // 94.000000

    // same case on local variable

    mut data\_local :=  [[f64(11),12,13]

             ,[f64(21),22,23]

             ,[f64(31),32,33]

             ,[f64(41),42,43]]

    data\_local << [[f64(98),97,96,95,94,93]]

    println(data\_local[4][4]) // 94.000000

}

In the following example (*struct\_address.v*) we see how to create a function that returns a struct:

Structs and global (module) space:

Using structs you can define const struct values for use in your program like this (see ch 13 - word\_counter\_book.v):

struct Mode {

    name string

    cli\_args []string

}

const (

    words = Mode{name: "words", cli\_args: ["-w", "--words"]}

    lines = Mode{cli\_args: ["-l", "--lines"], name: "lines"}

    chars = Mode{name: "chars", cli\_args: ["-c", "--chars"]}

)

Access modifiers - Visibility and mutability of fields: ??

Struct fields are private and immutable by default. Their access modifiers can be changed with pub and mut.

mut / pub / pub mut / \_\_global: see docs

struct Foo {

a int

mut:

b int

c int

pub:

d int

pub mut:

e int

\_\_global:

f int

}

?? example that shows errors

struct User {

email string

mut:

first\_name string

last\_name string

pub:

sin\_number int

pub mut:

phone int

\_\_global:

address\_1 string

address\_2 string

city string

country string

zip string

}

Struct allocation:

Structs are *value types*, they are by default allocated on the stack.

The following snippet creates 2 structs a and b, which are both *allocated on the stack*.

Listing 10.2B – alloc\_struct.v:

struct Aex {

  id int

}

fn new\_aex() Aex {

  return Aex{}      // allocated on the stack

  // return &Aex{}  // does a heap allocation

}

a := Aex{}

b := new\_aex()

A typical example of a struct is a time-interval, with a start- and end time (expressed here in seconds):

struct Interval {

start int

end int

}

(see interval.v)

Here are some initializations:

inter := Interval{start: 0, end: 3}

inter2 := Interval{end: 3, start: 0}

inter3 := Interval{start: 0}

inter4 := Interval{end: 3}

Because the fields are named the sequence must not be the same as in the declaration, and fields could also be omitted, like in the last cases.

Memory layout of a struct literal:

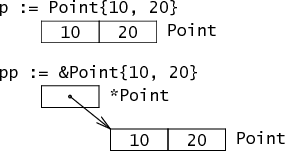


Figure 10.1: Memory layout of a struct

(\*Point should be &Point)

Structs of structs:

Structs in V and the data they contain, even when a struct contains other structs, form a continuous block in memory: this gives a huge performance benefit. This is unlike in Java with its reference types, where an object and its contained objects can be in different parts of memory; in V this is also the case with pointers.

This is clearly illustrated in the following example:

struct Rect1 { min Point, max Point }

struct Rect2 { min \*Point, max \*Point }

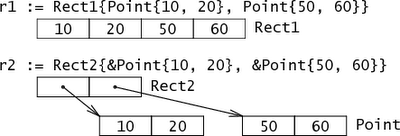


Figure 10.2: Memory layout of a struct of structs

Recursive structs:

A struct type can be defined in terms of itself. This is particularly useful when the struct variable is an element of a linked list or a binary tree, commonly called a *node.* In that case the node contains links (the addresses) to the neighboring nodes; next for a list and le, ri for a tree are pointers to another Node-variable.

data next

head next

tail nil

*Linked list*:

Figure 10.3: Linked list as recursive struct

where the data field contains the useful information (for example a f64), and next points to the successor node;

in V-code: struct Node {

data f64

next &Node

}

The first element of the list is called the *head*, it points to the 2nd element; the last element is called the *tail,* it doesn’t point to any successor, so its next field has value nil (??). In a real list we would have many data-nodes, the list can grow or shrink dynamically.

In the same way you could define a *doubly linked list* with a predecessor node field pr and a successor field su.

struct Node {

pr &Node

data f64

su &Node

}

*Binary tree:*

le data ri S

le data ri S

le data ri S

nil data nil S

Figure 10.4: Binary tree as a recursive struct

Here every node can at most have links to two other nodes: the left (le) and the right (ri); both of them can propagate this further. The top element of the tree is called the *root*; the bottom layer of nodes which have no more nodes beneath them are called the *leaves;* a leave node has (??) nil-values for the le- and ri pointers. Such a node is a tree in itself, so we could write:

in V-code: struct BinTree {

le &BinTree

data f64

ri &BinTree

}

From Alex (Jan 4 20):

struct Node {

left ?Node

right ?Node

}

Here is a schema on how to do a Depth-first traversing of a binary tree (call a function f on every node of binary tree bt, in depth-first infix order):

struct BinTree {

      le    &BinTree

      data  f64

      ri    &BinTree

}

fn (bt &BinTree) dfs(f fn(&BinTree)) {

  bt.le.dfs(f)

  f(bt)

  bt.ri.dfs(f)

}

Array of structs values:

A number of struct values of the same struct type can be put together in an array: (see user\_struct.v)

Generic structs: ?? example

Exercise:

Define a struct Person with fields name and gender. Make two struct values, and print them out. Make an array with these values and print out the array: see persons.v

## 9.2 Using a struct for memoization

When doing heavy computations one thing that can be done for increasing performance is not to repeat any calculation that has already been done, and reuse that. Instead cache the calculated value in memory, which is called *memoization*.

A great example of this is the Fibonacci program (see § 6.6): to calculate the n-th Fibonacci number, you need the 2 preceding ones, which normally have already been calculated.

If you do not store the preceding results, every higher Fibonacci number results in an ever greater avalanche of recalculations, which is precisely what the version from listing 6.13 (fibonacci.v) does.

Simple stock the n-th Fibonacci number in an array at index n (see chapter 7), and before calculating a Fibonaci-number, first look in the array if it has not yet been calculated. We see here clearly *how a struct value is injected into the function as a parameter, thus avoiding the need for a global variable.*

This principle is applied in the following listing (fibonacci\_memoized.v). The performance gain is astounding, time both programs for the calculation up to the 40th Fibonacci number.

struct Cache {

  mut:

    values []int

}

fn fib\_cached(n int, cache mut Cache) int {

  is\_in\_cache := cache.values.len > n

  if is\_in\_cache {

    return cache.values[n]

  }

  fib\_n := if n <= 1 {

    1

  } else {

    fib\_cached(n - 1, mut cache) + fib\_cached(n - 2, mut cache)

  }

  cache.values << fib\_n

  return fib\_n

}

mut c := Cache{}

for i := 0; i < 40; i++ {

  fib\_i := fib\_cached(i, mut c)

  print('$fib\_i - ')

}

/\*

1 - 1 - 2 - 3 - 5 - 8 - 13 - 21 - 34 - 55 - 89 - 144 - 233 - 377 - 610 - 987 - 1597 - 2584 - 4181

- 6765 - 10946 - 17711 - 28657 - 46368 - 75025 - 121393 - 196418 - 317811 - 514229 - 832040 - 1346269

- 2178309 - 3524578 - 5702887 - 9227465 - 14930352 - 24157817 - 39088169 - 63245986 - 102334155 -

\*/

For example, on a on Linux machine: *time ./fibonacci*

normal (fibonacci.v): the calculation took: 4.730270 s

with memoization: the calculation: 0.001000 s

In this algorithm memoization is obvious, but it can often be applied in other computations as well, perhaps using maps (see chapter 8) instead of arrays or slices.

Memoization is useful for relatively expensive functions (not necessarily recursive as in the example) that are called lots of times with the same arguments. It can also only be applied to pure functions, these are functions that always produce the same result with the same arguments, and have no side-effects.

Exercise: Apply the same mechanism to the factorial algorithm, see *factorial\_memoization.v*

## 9.3 Updating fields with the | operator

A new struct value can be made from an existing one, by changing one of its fields with the pipe operator |:

Listing update\_with\_pipe0.v:

struct User {

  name string

  age  int

}

fn main() {

  user1 := User{'Bob', 20}

  user2 := **{user1 | name: 'Peter'}** // to read as: struct user1, where name is modified to Peter

  println(user1.name)

  println(user2)

}

/\* Output:

Bob

{

  name: Peter

  age: 20

}

\*/

Here is an example where the update is done in a function register\_user, that takes a User struct value as argument (see listing update\_with\_pipe.v):

struct User {

  is\_registered bool

}

fn register\_user(u User) User {

  return **{u | is\_registered: true}**

}

fn main() {

  mut u := User{}

  u = register\_user(u)

  println(u.is\_registered.str()) // => true

}

In short: new\_foo := { foo | bar: baz } is the same as:

mut new\_foo := foo

new\_foo.bar = baz

## 9.4 Structs and pointers

Here is an example of changing struct field values through pointers (try to predict the output yourself):

Listing 4.22B – mut\_pointer.v:

struct S1 {

mut:

  i int

}

fn main() {

  mut p1 := &S1{}     // type: \*S1

  p1.i++

  mut p2 := p1        // type: mut(\*)S1 ??

  s1 := S1{}

  p2 = &s1

  p2.i++

  println(s1)

  s2 := S1{}

  p2 = &s2

  mut p3 := &s2       // type: \*mut(S1) ??

  p3.i++

  mut p4 := &s2

  p4.i++

  println(s2)

}

Here is an example of a struct with a reference field:

Listing 4.23 – struct\_pointer.v:

struct Sptr {

  data int

  ptr  &int

}

fn main() {

  s1 := Sptr{}

  s2 := Sptr{data: 5}

  println(s1)

  println(s2)

}

/\* Output:

{

  data: 0

  ptr: 0000000000000000

}

{

  data: 5

  ptr: 0000000000000000

}

\*/

Notice how the references are initialized with zeros, the null value does not exist in V!

A much more concrete example, also showing how to handle nil or null pointers (these are points with address 0): *rp\_game.v*

struct Place {

    name string

mut:

    left &Place

    right &Place

    previous &Place

}

struct Traveler {

mut:

    location &Place

}

fn nonnil\_or\_stay(old\_place &Place, new\_place &Place) &Place {

**if isnil**(new\_place) {

       println("Nothing is there.")

       return old\_place

    } else {

       return new\_place

    }

}

fn move(trav mut Traveler, direction string) {

    place := trav.location

    back := place.previous

    left := place.left

    right := place.right

    println("Old location $trav.location.name")

    println("Trying to move $direction")

    if direction == "back" {

        trav.location = nonnil\_or\_stay(place, back)

    } else if direction == "left" {

        trav.location = nonnil\_or\_stay(place, left)

    } else if direction == "right" {

        trav.location = nonnil\_or\_stay(place, right)

    } else {

        println("$direction is not valid, use back, left or right")

    }

    println("New location $trav.location.name")

}

fn main() {

    mut tree := Place{name: "Tree"}

    mut pile := Place{name: "Pile of old leaves"}

    mut shrub := Place{name: "Shrubbery"}

    mut bear := Place{name: "Bear behind Shrubbery"}

    // connect tree node with its children

    tree.left = &pile

    tree.right = &shrub

    pile.previous = &tree

    shrub.previous = &tree

    // forward-connect shrub node only

    // because it already has 'previous' set

    shrub.right = &bear

    bear.previous = &shrub

    // println(&tree)

    // println(tree)

    // println(&pile)

    // println(pile)

    // println(&shrub)

    // println(shrub)

    // println(&bear)

    // println(bear)

    traveler := Traveler{location: &tree}

    println(traveler.location.name) // Tree

    move(mut traveler, "back")

    move(mut traveler, "left")

    move(mut traveler, "back")

    move(mut traveler, "right")

    move(mut traveler, "back")

}

/\*

Tree

Old location Tree

Trying to move back

Nothing is there.

New location Tree

Old location Tree

Trying to move left

New location Pile of old leaves

Old location Pile of old leaves

Trying to move back

New location Tree

Old location Tree

Trying to move right

New location Shrubbery

Old location Shrubbery

Trying to move back

New location Tree

\*/

Output of the println:

000000000061FCF0

{

  name: Tree

  left: 000000000061FCC0

  right: 000000000061FC90

  previous: 0000000000000000

}

000000000061FCC0

{

  name: Pile of old leaves

  left: 0000000000000000

  right: 0000000000000000

  previous: 000000000061FCF0

}

000000000061FC90

{

  name: Shrubbery

  left: 0000000000000000

  right: 000000000061FC60

  previous: 000000000061FCF0

}

000000000061FC60

{

  name: Bear behind Shrubbery

  left: 0000000000000000

  right: 0000000000000000

  previous: 000000000061FC90

}

To cast voidptr to a V reference use:  user := &User(user\_void\_ptr)

voidptr can also be dereferenced to V structs by casting: user := User(user\_void\_ptr)

## 9.5 Methods

Let’s start with an example: Listing 10.3 – methods.v taken from the V docs:

struct User {

  age int

}

**fn** **(u User) can\_register() bool** {

  return u.age > 16

}

user := User{age: 10}

println(user.can\_register()) // => false

user2 := User{age: 20}

println(user2.can\_register()) // => true

The can\_register() function doesn’t take a struct value as a regular argument, but it has a u User as a *receiver argument*.

It is called on the struct value u as user.can\_register(), just like a method call in OO-languages.

PERF: Here u can be either passed by value (User) or by reference (&User). The compiler will make the right decision depending on the size of the User struct, which is different in Go where you have to make that decision yourself. In V you no longer have to remember which one to use. It works here because u can't be modified (it's not marked as `mut`).

### 9.5.1 What is a method?

Structs look like a simple form of classes, so an OO programmer might ask: where are the methods of the class? Again V has a concept with the same name and roughly the same meaning: a V *method* *is a function that acts on a variable of a certain type*, called the *receiver.* So a method is a special kind of function, one which has a receiver argument. A method and the type on which it acts must be defined in the same module, that’s why you cannot define methods on type int, float or the like.

The receiver type can be (almost) anything, not only a struct type: any type can have methods, even a function type or alias types for int, bool, string or array. But the receiver cannot be an interface type (§ 9.7), since an interface is the abstract definition and a method is the implementation. Lastly it cannot be a pointer type, but it can be a pointer to any of the allowed types.

The combination of a (struct) type and its methods is the V equivalent of a class in OO. One important difference is that the code for the type and the methods binding to it are not grouped together; they can exist in different source files, the only requirement is that they have to be in the same module.

The collection of all the methods on a given type T (or &T) is called the *method set* of T (or &T).

Methods are functions, so again there is *no method overloading:* for a given type, there is only one method with a given name.

(??) But based on the receiver type, there is overloading: a method with the same name can exist on 2 of more different receiver types, e.g. this is allowed in the same package:

func (a denseMatrix) add(b Matrix) Matrix

func (a sparseMatrix) add(b Matrix) Matrix

Also an alias of a certain type doesn’t have the methods defined on that type.

The general format of a method is:

fn (recv receiver\_type) method1(parameter\_list) (return\_value\_list) { … }

The receiver appears in its own argument list between the fn keyword and the method name. If recv is the receiver value and method1 the method name, then the call or invocation of the method follows the traditional object.method selector notation: **recv.method1()**

If in this expression, recv is a pointer, then it is automatically dereferenced.

If the method does not need to use the value recv, you can discard it by substituting a \_ , as in:

func (**\_** receiver\_type) method1(parameter\_list) (return\_value\_list) { … }

recv is like the this- or self from OO-languages, but in V there is no specified keyword for it.

STYLE: The convention is not to use receiver names like self or this, but a short, preferably one letter long, name.

In the following example (*structs.v*), we see:

* How struct values can be constants in line (1)
* A kind of factory function new\_address for creating a new struct value in line (2)
* A method str() on struct Address for a customized output in line (3)
* - How a constant can be calculated by calling a function in line (4)

struct Address {

  street string

  city string

  state string

  zip int

}

const (

    streets = ['1234 Alpha Avenue', '9876 Test Lane']

    test\_address = Address {street : streets[0], city: 'Beta', state : 'Gamma', zip : 31416} (1)

    test\_address2 = Address {street : streets[1], city: 'Exam', state : 'Quiz', zip : 62832}

)

fn new\_address(street, city, state string, zip int) Address { (2)

    return Address{street : street, city : city, state : state, zip : zip}

}

fn (a Address) str() string { (3)

    return 'Address.str(): $a.street, $a.city, $a.state $a.zip'

}

const (

    address1 = new\_address('2718 Tau Dr', 'Turing', 'Leibniz', 54366) (4)

    address2 = new\_address('3142 Uat Rd', 'Einstein', 'Maxwell', 62840)

)

println(streets)

println('$test\_address.street, $test\_address.city, $test\_address.state $test\_address.zip')

println('$test\_address2.street, $test\_address2.city, $test\_address2.state')

println(address1.str())

println(address2.str())

/\* Output:

["1234 Alpha Avenue", "9876 Test Lane"]

1234 Alpha Avenue, Beta, Gamma 31416

9876 Test Lane, Exam, Quiz

Address.str(): 2718 Tau Dr, Turing, Leibniz 54366

Address.str(): 3142 Uat Rd, Einstein, Maxwell 62840

\*/

Here is an example of a method that returns 3 values (see *point\_methods.v):*

You can see that the method test\_out\_of\_order\_calls can call method dist in line (1) before it is defined in line (2)

import math

struct Point {

    x int

    y int

}

fn test\_out\_of\_order\_calls() {

    point := Point{x : 2, y : 2}

    mut point1 := Point{}

    point1 = Point{x : 1, y : 1}

    x\_diff, y\_diff, distance := point.dist(point1) (1)

    println('The x distance is: $x\_diff')

    println('The y distance is: $y\_diff')

    println('The distance between the two points is ${distance:.2f}')

}

fn (p Point) dist(p2 Point) (f64, f64, f64)  {   (2)

    mut x\_diff := f64(p2.x - p.x)

    mut y\_diff := f64(p2.y - p.y)

    x\_diff = math.pow(x\_diff, 2)

    y\_diff = math.pow(y\_diff, 2)

    distance := math.sqrt(x\_diff + y\_diff)

    return x\_diff, y\_diff, distance

}

/\* Output:

The x distance is: 1.000000

The y distance is: 1.000000

The distance between the two points is 1.41

\*/

test\_out\_of\_order\_calls()

#### Mutable receivers

For a receiver to be changeable, it must be indicated as mut, and it will always be passed as a reference.

Here is the equivalent code of update\_with\_pipe.v from § 9.4, but now with a method instead of a normal function (see listing 10.3 - mutable\_receiver.v):

struct User {

mut:

    is\_registered bool

}

fn **(u mut User)** register() {

    u.is\_registered = true

}

mut user := User{}

println(user.is\_registered) // => false

user.register()

println(user.is\_registered) // => true

println(user)

/\*

false

true

{

  is\_registered: 1

}

\*/

The register() function has a u mut User as a *receiver argument*. It is called on the struct value u as user.register(), just like a method call in OO-languages. Mutable receivers that are not modified result in a compilation error.

user is of type User, but it can be treated as pointer type and it is implicitly converted (like in Go),so you write mut user instead of &user.

In the following example (see *mutable\_receiver2.v*), you see that a mutable receiver is in fact passed by reference:

struct Item {

mut:

  thing int

}

fn (i mut Item) do\_thing() Item {

  i.thing = 1

  return \*i

}

mut itm := Item{}

itm.do\_thing()

println(itm)

/\* Output:

{

  thing: 1

}

\*/

#### A struct cannot contain a field of its type:

struct Item {

other Item

}

This gives the error message: cannot embed struct `Item` in itself (field `other`)

But it can contain a reference to itself:

struct Item {

other &Item

}

compiles and can be used in code, as is clear from the following example:

#### A struct with a reference field to itself: see ref\_field.v

struct Item {

  s     string

  other &Item

}

fn main() {

  i\_2 := &Item{s: 'Hello, '}

  i\_1 := Item{s: 'World!', other: i\_2}

  println(i\_1.other.s + i\_1.s) // Hello, World!

}

#### A struct can contain a field with its type:

struct Item {

other ?Item

}

#### The str() method

Whenever println is called on a type, it uses the str() method on that type to stringify it, and then show that value. If there is no str() method available, a compiler error is given.

Structs have a default str() method, so that they can be printed out. Here is an example where the Color struct implements this method to have a more compact representation:

Listing 10.3C – color\_struct.v:

struct Color {

        r int

        g int

        b int

}

pub fn (c Color) str() string { return '{$c.r, $c.g, $c.b}' }

fn rgb(r, g, b int) Color { return Color{r: r, g: g, b: b} }

const (

        red  = Color{r: 255, g: 0, b: 0}

        blue = rgb(0, 0, 255)

)

println(red)    // => {255, 0, 0}

println(blue)   // => {0, 0, 255}

It also shows an example of a constant struct value (red), and even a constant blue which is calculated at compile-time by calling the rgb function

(the Color struct is defined in the gx module).

#### Struct containing an array of structs – Use of the for in loop:

In the following example we have a str() method for the User struct that only prints the user’s name. Then we create an array users with two user structs. We iterate over this array to print out all names with for in:

Listing 10.3B – structs.v

struct User {

  id string

  name string

}

fn (u User) str() string {

  return u.name

}

fn main() {

  users := [

   User{id: '01', name: 'John'},

   User{id: '02', name: 'Amy'},

  ]

  for user in users {

    println(user)

  }

}

/\* Output:

John

Amy

\*/

Here is a slightly more complicated example containing these features: a struct Repo containing an array of User structs. We have made our own custom str() method to print out the repo value. We also see a function new\_repo(), that returns a struct value.

Listing 10.3C – forin\_struct.v:

struct User {

        id int

name string

age int

}

struct Repo {

mut:

**users []User**

}

fn new\_repo() Repo {

user := User{id:10, name: 'Bill', age: 45}

return Repo {

users: [user]

}

}

pub fn **(r Repo) str() string** {

        mut str := ''

        for u in r.users {

                str += '$u.id : $u.name / $u.age\n'

        }

        return str

}

mut repo := new\_repo()

user1 := User{id: 12, name: 'Denise', age: 60}

user2 := User{}

user3 := User{id: 42, name: 'Linda', age: 45}

repo.users << user1

repo.users << user2

repo.users << user3

**for u in repo.users** {

    println('$u.id - $u.name: $u.age')

}

println('')

// using the str() method for Repo:

println(repo)

/\* Output:

10 - Bill: 45

12 - Denise: 60

0 - : 0

42 - Linda: 45

10 : Bill / 45

12 : Denise / 60

0 :  / 0

42 : Linda / 45

\*/

The following example Listing 10.4 – person.v shows a struct Person, a method up\_person which has a parameter of type mut Person (so that the object itself can be changed) and 3 different ways of calling this method:

struct Person {

mut:

        first\_name string

        last\_name  string

}

fn (p mut Person) up\_person() {

        p.first\_name = p.first\_name.to\_upper()

        p.last\_name = p.last\_name.to\_upper()

}

// 1- struct as a value type:

mut pers1 := Person{}

pers1.first\_name = 'Chris'

pers1.last\_name = 'Woodward'

pers1.up\_person()

println('The name of the person is $pers1.first\_name $pers1.last\_name')

// 2 - struct as a literal, created on the stack:

mut pers2 := Person{first\_name: 'Chris', last\_name: 'Woodward'}

pers2.up\_person()

println('The name of the person is $pers2.first\_name $pers2.last\_name')

// 3 - struct as a literal, created on the heap:

mut pers3 := &Person{first\_name: 'Chris', last\_name: 'Woodward'}

pers3.up\_person()

println('The name of the person is $pers3.first\_name $pers3.last\_name')

/\* Output:

The name of the person is CHRIS WOODWARD

The name of the person is CHRIS WOODWARD

The name of the person is CHRIS WOODWARD

\*/

**EXERCISES:**

 Create a struct Animal, give it a field to specify that it has fur or not: *has\_fur.v*

Exercise 10.1: rectangle.v:

Define a struct Rectangle with int properties length and width. Give this type the methods area() and perimeter() and test it out.

Exercise 10.2: employee\_salary.v Define a struct Employee with a field salary, and a method give\_raise to increase the salary with a certain percentage.

 Create a method for the type Person that determines whether a person is underage or not: *under\_age.v*

### 9.5.2 Difference between a function and a method

A function has the variable recv as a parameter: function1(recv)

A method is called on the variable recv: recv.method1()

A method can change the values (or the state) of the receiver variable provided this is indicated as mut, just as is the case with functions (a function can also change the state of its parameter when this is passed as mut).

!! Don’t forget the ( ) after method1, or you get the compiler error: expected `(`

The receiver must have an explicit name, and this name must be used in the method.

receiver\_type is called the *(receiver) base type*, this type must be declared within the same module as all of its methods.

In V the methods attached to a (receiver) type are not written inside of the structure, as is the case with classes. The coupling is much more loose: the association between method and type is established by the receiver.

*Methods are not mixed with the data definition (the structs): they are orthogonal to types.*

*Representation (data) and behavior (methods) are independent.*

### **9.5.3. Operator overloading**

(see operator\_overloading.v)

Operator overloading goes against V’s philosophy of simplicity and predictability. But since scientific and graphical applications are among V’s domains, operator overloading is very important to have in order to improve readability:

a.add(b).add(c.mul(d)) is a lot less readable than a + b + c \* d.

To improve safety and maintainability, operator overloading has several limitations:

- It’s only possible to overload +, -, \*, / operators.

- Calling other functions inside operator functions is not allowed.

- Operator functions can’t modify their arguments.

- Both arguments must have the same type (just like with all operators in V).

Example:

The following code defines a + and – operator for a vector struct Vec:

struct Vec {

  x int

  y int

}

fn (a Vec) str() string {

  return '{$a.x, $a.y}'

}

fn **(a Vec) + (b Vec)** Vec {

  return Vec {

    a.x + b.x,

    a.y + b.y

  }

}

fn **(a Vec) - (b Vec)** Vec {

  return Vec {

    a.x - b.x,

    a.y - b.y

  }

}

fn main() {

  a := Vec{2, 3}

  b := Vec{4, 5}

  println(a + b) // "{6, 8}"

  println(a - b) // "{-2, -2}"

}

See also the overloading of normal number operations in module math.complex:

// Complex Addition c1 + c2

pub fn (c1 Complex) + (c2 Complex) Complex {

return Complex{c1.re + c2.re, c1.im + c2.im}

}

## 9.6 Interfaces

V is not a ‘classic’ OO language: it doesn’t know the concept of classes and inheritance.

However it does contain the very flexible concept of *interfaces*, with which a lot of aspects of object-orientation can be made available. Interfaces in V provide a way to *specify the behavior* of an object: if something can do this, then it can be used here.

An interface defines a set of methods (the *method set*), but these methods do not contain code: they are not implemented *(* they are *abstract)*. Also an interface cannot contain variables.

An interface is declared in the format: **interface** Namer {

method1(param\_list) return\_type

method2(param\_list) return\_type

…

}

Namer is an *interface type.*

The name of an interface is formed by the method name plus the [e]r suffix, such as Printer, Reader, Writer, Logger, Converter, and so on, thereby giving an active noun as a name. A less used alternative (when ..er is not so appropriate) is to end it with able like in Recoverable, or to start it with an I (more like in .NET or Java) .

Interfaces in V are short, they usually have from 1 – max 3 methods.

Types (like structs) can have the method set of the interface implemented; the implementation contains for each method real code how to act on a variable of that type: *they implement the interface*, the method set forms the interface of that type.

*A type doesn’t have to state explicitly that it implements an interface: interfaces are satisfied implicitly.*

*Multiple types can implement the same interface.*

*A type that implements an interface can also have other functions.*

*A type can implement many interfaces.*

*An interface type can contain a reference to an instance of any of the types that implement the interface.*

Even if the interface was defined later than the type, in a different module, compiled separately: if the object implements the methods named in the interface, then it implements the interface.

All these properties allow for a lot of flexibility.

As a first example, look at Listing 11.1A – interface\_point.v:

struct Point {

  x i8

  y i8

  z i8

}

fn (p Point) draw() string {

  return 'Point(${p.x},${p.y})'

}

fn to\_string(d Drawer) string {

  return d.draw()

}

interface Drawer {

  draw() string

}

p := Point{x: 2, y: 3}

println(to\_string(p)) // Point(2,3)

Point implements the interface Drawer, because Point has a method draw(). That’s why a Point value can be used in any case where a Drawer value is needed, like in the to\_string method.

As a second example, look at Listing 11.1B – house\_animals.v: (from the V Docs)

struct Dog { }

struct Cat { }

fn (d Dog) speak() string {

  return 'woof'

}

fn (c Cat) speak() string {

  return 'meow'

}

interface Speaker {

speak() string

}

fn perform(s Speaker) string {

   return s.speak()

}

dog := Dog{}

cat := Cat{}

println(perform(dog)) // -> "woof"

println(perform(cat)) // -> "meow"

Both the Dog and Cat struct type implement the Speaker interface, because they both have a speak() method with the same signature as that of the interface.

As a third example, look at Listing 11.1C - interfaces.v:

interface Shaper {

        area() f64

// perimeter() f64

}

struct Square {

mut:

         side f64

}

fn (sq Square) area() f64 {

         return sq.side \* sq.side

}

fn calc\_area(sh Shaper) f64 {

return sh.area()

}

mut sq1 := Square{}

sq1.side = 5

println('The square has area: ${calc\_area(sq1)}')

Output: The square has area: 25.000000

The program defines a struct Square and an interface Shaper, with one method area().

In main() a value of Square is constructed. Outside of main we have an area() method with a receiver type of Square where the area of a square is calculated: the struct Square implements the interface Shaper.

Because of this we can assign a variable of type Square to a variable of the interface type Shaper when calling the calc\_area function. Now the interface variable contains a reference to the Square variable and through it we can call the method area() on Square. Of course you could call the method immediately on the Square value sq1.area(), but the novel thing is that we can call it on the interface value, thereby generalizing the call.

This is V’s version of *polymorphism*, a well known concept in OO software: the right method is chosen according to the current type, or put otherwise: a type seems to exhibit different behaviors when linked to different values.

If Square would not have an implementation of area(), we would receive the compiler error:

Type "Square" doesnt satisfy interface "Shaper" (method "area" is not implemented)

The same error would occur if Shaper had another method perimeter(), and Square would not have an implementation for that, even if perimeter() was not called on a Square value.

duck\_dance.v doesn’t work (Nov 11) ??

Arrays of interfaces []interface are supported.

How does it work? (Reddit 18: Question about interfaces - to verify)

V interfaces are very efficient.

Dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to call at run time. In V there is no selection: a type does not have a vtable with function pointers to its methods.

Instead in V the compiler knows exactly what function is called, not just a function pointer or vtable.

A receiver type cannot be an interface:

interface Speaker {}

fn (c Speaker) speak() string {

return 'meow'

}

Gives the compiler error: invalid receiver type `Speaker` (`Speaker` is an interface)

Bottom of Form

?? Exercise: Expand the example with a type Rectangle which also implements Shaper. We can now make an array with elements of type Shaper, and show polymorphism in action by using a for range on it and calling Area() on each item. (see Listing 11.2 – interfaces\_poly.go:)

?? Exercise 11.2-3: interfaces\_poly2.go

1. Expand the example interfaces\_poly.go to a type Circle: interfaces\_poly2.go
2. Now we will implement the same functionality by using an ‘abstract’ type Shape (abstract because it has no fields) which implements Shaper, and embedding his type in the other types. Now demonstrate that overriding is used as explained in § 10.6.5: interfaces\_poly3.go

Perhaps you can now begin to see how interfaces can produce cleaner, simpler, and more scalable code.

An interface is a kind of contract which the implementing type(s) must fulfill.

Interfaces describe the behavior of types, what they can do. They completely separate the definition of what an object can do from how it does it, allowing distinct implementations to be represented at different times by the same interface variable, which is what polymorphism essentially is.

Writing functions so that they accept an interface variable as a parameter makes them more general.

!! Use interfaces to make your code more generally applicable !!

This is also ubiquitously applied in the code of the standard library. It is impossible to understand how it is build without a good grasp of the interface-concept (??)

## 9.7 Sum types

(see docs)

## 9.8 V and object orientation

### ?? 9.8.1 How to embed functionality in a type

V doesn't have subclassing, but it supports embedded structs.

*Aggregation* (or *composition*): include a *named* field of the type of the wanted functionality

Nov 9 : not yet supported: see embed\_func1.v

### ?? 9.8.2 Multiple inheritance

Apr 13 : not yet supported: see mult\_inheritance.v

### ?? 9.8.3 Comparison between V types and methods and other object-oriented languages.

Methods in OO languages like C++, Java, C# or Ruby are defined in the context of classes and inheritance: when a method is invoked on an object, the runtime sees whether its class ore any of its superclasses have a definition for that method, otherwise an exception results.

In V such an inheritance hierarchy is not at all needed: if the method is defined for that type it can be invoked, independent of whether or not the method exists for other types; so in that sense there is a greater flexibility.

This is nicely illustrated in the following schema (?? To change for V):

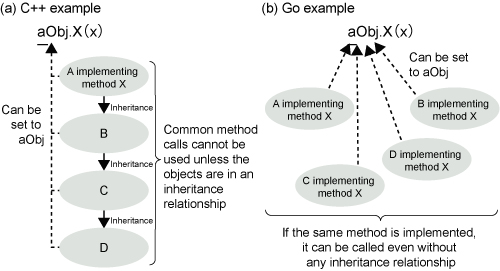


Figure 10.4: Methods in V and OO-languages

V doesn't require an explicit class definition as Java, C++, C#, etc do. Instead, a "class" is implicitly defined by providing a set of methods which operate on a common type. This type may be a struct or any other user-defined type.

In Java or C# the type and the func would be placed together in a class Integer; in Ruby you could just write the method on the basic type int.

Summarized: in V types are basically classes (data and associated methods). V doesn’t know inheritance like class oriented OO languages. Inheritance has two main benefits: code reuse and polymorphism.

Code reuse in V is achieved through composition and delegation, and polymorphism through the use of interfaces: it implements what is sometimes called *component programming*.

Many developers say that V’s interfaces provide a more powerful and yet simpler polymorphic behavior than class inheritance.

### 9.8.4 Summary: the object-orientedness of V

Let us summarize what we have seen about this: V has no classes, but instead loosely coupled types and their methods, in some cases implementing interfaces.

The 3 important aspects of OO-languages are encapsulation, inheritance and polymorphism, how are they realized in V?

i) Encapsulation (data hiding): see § 9*.3*

All data is local, private and immutable by default. There are no global variables.

In order for data to be visible outside a module, it must be declared pub(lic).

A type can only have methods defined in its own module.

ii) Inheritance: how? Through composition: embedding of 1 (or more) type(s) with the desired behavior (fields and methods); multiple inheritance is possible through embedding multiple types

iii) Polymorphism: how? Through interfaces: a variable of a type can be assigned to a variable of any interface it implements. Types and interfaces are loosely coupled, multiple inheritance is possible through implementing multiple interfaces. V's interfaces aren't a variant on Java or C# interfaces, they're much more: they are independent and are key to large-scale programming and adaptable, evolutionary design.

So in effect we can say that V implements all important object-oriented features.