# Chapter 10 –Option/Result Types and Error Handling

## 10.1 What is an option type?

In V no value can be nil or null, all values are initialized to non-nil values.

A function that returns a value will in most cases return a normal value, but in an extra-ordinary situation it could return

an error, in case something went wrong

or a none value, in case a value cannot be found.

In V such a return value is represented as an o*ption type*. An option value is either a normal value, or an abnormal value (an error or none); only one of both is valid at any time. If the value is of type T, then the option type is written as: ?Type. (read it as optional Type).

So for example an optional int is written as ?int, a generic optional type can be written as ?T

This *option* type in V resembles the optional type in Swift, C# and Kotlin, and it combines the features of the Option<T> and Result<T> types in Rust. So you as a developer don’t need to decide which one to use!

Here is a simple example:

Listing 9.1 optional1.v:

fn f() **?int** {

a := 5

return a

}

fn main() {

o := **f() or {**

**return**

**}**

println(o) // => 5

}

The function f() returns an option int. Because this is just an education example, we make sure the functions returns the integer 5.

When f() is called, the int value is unwrapped and assigned to o.

If an error occurs or no value is found, the or block is executed.

*An option type must always be handled by an or block.*

If you forget the or block, the compiler gives the error: unhandled option type: ?int

The or unwraps the value and assigns it to the left-hand side variable, but when an error or none is returned, the code block after the or is executed. This block must handle the error or none value, and it must end with return, break, or continue.

The or block can process the error (pass it on, print it, log it, …) or similarly process the none value. The block must end with return, exit, panic, break, or continue.

(??) You can’t leave out the `main` function here, because main needs to return ``.

Let’s now look at a function that returns an optional struct:

Listing 9.2 optional2A.v:

struct User {

   id int

}

fn find\_user\_by\_id(id int) ?**User** {

u := User{id: 10}

return u

}

fn main() {

user := find\_user\_by\_id(10) **or {**

**return**

**}**

println(user.id)

}

The function find\_user\_by\_id returns an option ?User value. In this case the returned value is hard-coded, so there is no possibility of returning an error or none. Nevertheless, we again see the specific pattern that is used when retrieving an option value: the or code block (try out deleting it).

Here is a more realistic example: option2B.v

struct User {

  id int

  name string

}

struct Repo {

  users []User

}

fn new\_repo() Repo {

        return Repo {

                users: [User{1, 'Andrew'}, User {2, 'Bob'}, User {10, 'Charles'}]

        }

}

fn (r Repo) find\_user\_by\_id(id int) **?User** {

  for user in r.users {

    if user.id == id {

      // V automatically wraps this into an option type

**return user**

    }

  }

**return****error('User $id not found!')**

}

fn main() {

  repo := new\_repo()

  user := repo.find\_user\_by\_id(10) **or {**

**println(err)**

**return**

**}**

  println(user.id)   // => 10

  println(user.name) // => "Charles"

}

Here we see that in case a User with the specified id is not found, the error() function is called to pass a string error message: **error('User $id not found!')** .

Inside an or block the variable err is defined, which is set to the error string message to be printed out (using *eprint*ln might be better here to send the message to the standard error output sterr). err is empty if none was returned.

Here is an example of an error output:

user := repo.find\_user\_by\_id(999) or {

println(err) // => User 999 not found!

return

}

While coding you’ll probably define a function as returning a certain type at the first pass, and then in a second pass realize that an option type is needed.

The amount of additional work required to "upgrade" a function to a function returning an option type is minimal: you have to prefix  the return type with ? and return an error when something goes wrong.

Here is an example where a division by zero error is handled in V (see *option3.v)*:

struct DivisionResult {

    result f64

}

fn divide(a, b f64) **?DivisionResult** {

    if (b != 0) {

        return DivisionResult {result : a/b }

    }

    return **error('Can\'t divide by zero!')**

}

fn error\_handling\_example() {

    x := f64(10.0)

    y := f64(0)

    z := f64(2.5)

    succeed := divide(x, z) or {

        println(err)

        return

    }

    println(succeed.result) // => 4.000000

    fail := divide(x, y) or {

        println(err) // => Can't divide by zero!

        return

    }

    println(fail.result) // code not reached

}

error\_handling\_example()

If syntax for handling optionals: (?? Example)

if a := func() { }

## 10.2 Error-handling

Working with option types and or blocks is the primary way of handling errors in V. They are still values like in Go, but the advantage is that *errors in V can't be unhandled, and handling them is a lot less verbose.*

### Comparing V and Go

// this is the way it works in Go:

result, err := unsafe\_function()

**if** err != nil {

// ... handle your error

}

// this is the way it works in V:

result := unsafe\_function() or {

// ... handle your error

}

### The error message and the err variable

An error value can be created with a message string, like this: error('User $id not found')

If the function that returns an option type finds itself in the error condition, you write: return error('error\_message')

This is caught in the or block in the calling function. The error-info is also stored in an err variable, so the or block could look like this:

or {

**eprintln**('Failed to fetch "users.json": **$err**')

return

}

err is a reserved variable (not a global) that contains an error message if there is one.

eprintln is a special version of println to print out error information. This can be a simple message, or a message in which $err is used.

### Propagating an error with ?

You can also use the ? operator to propagate the error to the calling function.

? passes the error on, if it happens in main(), the program panics (like in Rust, which uses the similar `unwrap` or the ! operator in Swift):

In main()

f := foo()?

Works exactly the same as:

f := foo() or { panic(err) }

The ? operator is written right after the function call that returns an option type.

Here is a concrete example:

resp := http.get(url)?

println(resp.body)

http.get returns the option type ?http.Response (if the url is not found, an error is returned).

It was called with ?, so the error is propagated to the calling function (which must return an optional) or in case of main leads to a panic.

So the code above is an even shorter version of:

resp := http.get(url) or {

panic(err)

}

println(resp.body)

?? For an example of error-handling when decoding JSON data, see § 13.2

If you are calling C functions and need to access their error codes, you can do this with C.errno.

?? V uses C’s atomic.

### Panic

We have often seen how the compiler generates a panic at runtime for example when dividing by zero, for example (see panic.v):

fn main(){

  println(9 / 0)

}

Which gives as output:

[Running] v run panic.v

**panic.v:2:14: division or modulo by zero**

    1| fn main(){

    2|  println(9 / 0)

                    ^

But you can also generate a runtime panic with a message yourself, as in this example:

fn main(){

**panic('I created a deliberate panic!')**

}

With output:

[Running] v run panic.v

**V panic: I created a deliberate panic!**

This can be useful as an indication that some improbable or otherwise unreachable code has executed. Time to review your code!