Passing arguments to programs

When we write a program, we often want to pass information to it. We can do this by providing arguments when we run the program from the command line.

This arguments are always strings. Even if we write numbers they reach the program as strings. If we want to use them as numbers we need to convert them in our code (or use libraries that do it for us).

The most common way to send arguments is by preceding the value with a dash (`-`) or two dashes (`--`) and a letter or word and after that the value for the argument:

```
ls --long --all
```

Here `ls` gets two arguments: `--long` and `--all`. Some programs also allow short arguments with a single dash and a single letter:

```
ls -l -a
```

```
pandoc -s -o output.pdf input.md
```

Here `pandoc` gets three arguments: `-s`, `-o` and `output.pdf`. The last argument is not preceded by a dash, so it is considered a positional argument. `-o output.pdf` is a flag with a value, it tells the program that the output should go to a file named `output.pdf`.

The same option or flag is used in `go build` too:

```
go build -o myprogram.exe main.go
```

Accessing arguments in Go

There are two main ways to access arguments in Go: using the `os` package or using the `flag` package.

Using the `os` package

The `os` package provides a platform-independent interface to operating system functionality.

Through the variable `os.Args`, provided by the package, we can access all the arguments passed to a Go program. This variable is a slice of strings that contains all the arguments, including the program name itself as the first element.

```
package main

import (
    "fmt"
    "os"
)

func main() {
    // os.Args is a slice of strings that contains all the arguments passed to the program
```

```
    args := os.Args

    if len(args) == 1 { // os.Args always contains at least the program name.
        fmt.Println("No arguments provided")
        return
    }

    for i, arg := range args {
        fmt.Printf("Argument %d: %s\n", i, arg)
    }
}
```

As you can see, the `os.Args` does not parse the arguments for us. All we get is a slice of strings. If we want to interpret them as flags or options, we need to do it ourselves.

Parsing arguments manually

To parse a string to other types, we can use the `strconv` package. `strconv.Atoi` converts a string to an integer, `strconv.ParseBool` converts a string to a boolean, etc.

With an example, let's say we want to parse two arguments: `-a` which is an integer and `--name` which is a string.

```
package main

import (
    "fmt"
    "os"
    "strconv"
)

func main() {
    args := os.Args

    if len(args) < 3 {
        fmt.Println("Usage: myprogram -a <number> --name <string>")
        return
    }

    var a int
    var name string

    for _, arg := range args {
        if arg == "-a" && len(args) > 2 {
            a, err := strconv.Atoi(args[2]) // Convert string to int
            if err != nil {
                fmt.Println("Error:", err)
                fmt.Println("Invalid value for -a:", args[2])
                return
            }
        } else if arg == "--name" && len(args) > 4 {
            name = args[4]
        }
```

```
    }

    fmt.Printf("Value of -a: %d\n", a)
    fmt.Printf("Value of --name: %s\n", name)
}
```

The function `strconv.Atoi` returns an error if the conversion fails, so we need to handle that case:

```
x, err := strconv.Atoi("notanumber")

if err != nil {
    fmt.Println("Error:", err)
} else {
    fmt.Println("Converted value:", x)
}
```

This is the way we use to handle errors in Go. If the function returns an error, we check if it is not `nil` and handle it accordingly.

Using the `flag` package

The `flag` package provides a more convenient way to parse command-line arguments. It allows us to define flags and their types, and it automatically handles parsing and error checking.

To use the `flag` package, we need to import it and define our flags using the `flag.Int`, `flag.String`, etc. functions. After defining the flags, we call `flag.Parse()` to parse the command-line arguments.

The functions to define flags have the following signature `flag.Type(name string, defaultValue Type, usage string) *Type` where `Type` can be `Int`, `String`, `Bool`, etc. The function returns a pointer to the value of the flag.

Here, the arguments of the functions are:

- `name`: the name of the flag (without dashes).
- `defaultValue`: the default value of the flag if it is not provided.
- `usage`: a description of the flag that will be shown in the help message.

If the value passed to `name` is `all` this means that the flag can be used as `-all` or `--all`.

So, if we want to invoque our program like this:

```
myprogram -a 42 --name John
```

We will need to set up the flags like this:

```
age := flag.Int("a", 0, "an integer value")
name := flag.String("name", "default", "a string value")
```

There are two ways of get flags: using **pointers** or using **variables**.

Flags with pointers

The functions `Int`, `String`, etc. return pointers to the values of the flags. We need to **dereference the pointers** to get the actual values using the `*` operator.

```
package main

import (
    "flag"
    "fmt"
)

func main() {
    // Define flags using pointers.
    // `a` is a pointer to an int, to retrieve the value we need to dereference it (`*a`).
    a := flag.Int("a", 0, "an integer value")
    // `name` is a pointer to a string, to retrieve the value we need to dereference it
(`*name`).
    name := flag.String("name", "default", "a string value")

    // Parse the command-line arguments.
    flag.Parse()

    // Use the values of the flags.
    fmt.Printf("Value of -a: %d\n", *a)       // Dereference the pointer to get the value
    fmt.Printf("Value of --name: %s\n", *name) // Dereference the pointer to get the value
}
```

Flags with variables

There are alternative functions to `Int`, `IntVar`; `String`, `StringVar`, etc. that allow us to define flags using variables instead of pointers. We need to pass the address of the variable to the function using the `&` operator.

```
package main

import (
    "flag"
    "fmt"
)

func main() {
    var a int
    var name string

    // Define flags using variables.
    // We need to pass the address of the variable using the `&` operator so the flag
package can set the value directly.
    flag.IntVar(&a, "a", 0, "an integer value")
    flag.StringVar(&name, "name", "default", "a string value")

    // Parse the command-line arguments.
```

```
    // Be aware that flag.Parse() must be called after defining all flags.
    // If we don't call it won't be any error but the values of the flags will be the
default ones.
    flag.Parse()

    // Use the values of the flags directly.
    fmt.Printf("Value of -a: %d\n", a)       // No need to dereference, we have the value
directly
    fmt.Printf("Value of --name: %s\n", name) // No need to dereference, we have the value
directly
}
```

> We need to remember to call `flag.Parse()` after defining all the flags. If we don't call it, the values of the flags will be the default ones.> Someone didn't do it last time in class with underwhelming results.