```
    USER = os.Getenv("USER")
    GOROOT = os.Getenv("GOROOT")
)
```

The var syntax is mainly used at a global, package level, in functions it is replaced by the *short declaration syntax* := (see § 4.4).

Here is an example of a program which shows the operating system on which it runs. It has a local string variable getting its value by calling the Getenv function (which is used to obtain environment-variables) from the os-package.

Listing 4.5—goos.go:

```
package main
import (
    "fmt"
    "os"
)

func main() {
    var goos string = os.Getenv("GOOS")
    fmt.Printf("The operating system is: %s\n", goos)
    path := os.Getenv("PATH")
    fmt.Printf("Path is %s\n", path)
}
```

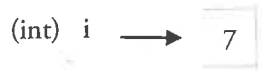
The output can for example be: The operating system is: windows, or The operating system is: linux, followed by the contents of the path variable.

Here Printf is used to format the ouput (see § 4.4.3).

4.4.2 Value types and reference types

Memory in a computer is used in programs as a enormous number of boxes (that's how we will draw them), called *words*. All words have the same length of 32 bits (4 bytes) or 64 bits (8 bytes), according to the processor and the operating system; all words are identified by their *memory address* (represented as a hexadecimal number).

All variables of elementary (primitive) types like int, float, bool, string, … are *value types*, they point directly to their value contained in memory:

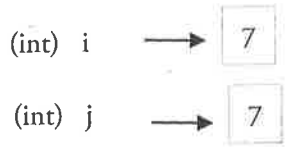


32 bit word

Fig 4.1: Value type

Also composite types like arrays (see chapter 7) and structs (see Chapter 10) are value types.

When assigning with = the value of a value type to another variable: `j = i`, a copy of the original value i is made in memory.



Assigns Value

Fig 4.2: Assignment of value types

The memory address of the word where variable i is stored is given by &i (see § 4.9), e.g. this could be 0xf840000040. Variables of value type are cointained in *stack* memory.

The actual value of the address will differ from machine to machine and even on different executions of the same program as each machine could have a different memory layout and and also the location where it is allocated could be different.

More complex data which usually needs several words are treated as reference types.
A *reference type* variable r1 contains the address (a number) of the memory location where the value of r1 is stored (or at least the 1st word of it):

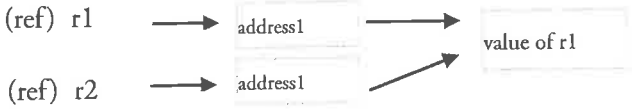


Fig 4.3: Reference types and assignment

This address which is called a *pointer* (as is clear from the drawing, see § 4.9 for more details) is also contained in a word.

The different words a reference type points to could be sequential memory addresses (the memory layout is said to be *contiguously*) which is the most efficient storage for computation; or the words could be spread around, each pointing to the next.

When assigning r2 = r1, only the reference (the address) is copied.

If the value of r1 is modified, all references of that value (like r1 and r2) then point to the modified content.

In Go pointers (see § 4.9) are reference types, as well as slices (ch 7), maps (ch 8) and channels (ch 13). The variables that are referenced are stored in the *heap*, which is garbage collected and which is a much larger memory space than the stack.

4.4.3 Printing

The function `Printf` is visible outside the fmt-package because it starts with a P, and is used to print output to the console. It generally uses a format-string as its first argument:

```
func Printf(format string, list of variables to be printed)
```

In Listing 4.5 the format string was: `"The operating system is: %s\n"`

This format-string can contain one or more format-specifiers %.., where .. denotes the type of the value to be inserted, *e.g.* **%s** stands for a string-value. **%v** is the general default format specifier. The value(s) come in the same order from the variables summed up after the comma, and they are separated by comma's if there is more than 1. These % placeholders provide for very fine control over the formatting.

The function `fmt.Sprintf` behaves in exactly the same way as Printf, but simply returns the formatted string: so this is the way to make strings containing variable values in your programs (for an example, see `Listing 15.4—simple_tcp_server.go`).

The functions `fmt.Print` and `fmt.Println` perform fully automatic formatting of their arguments using the format-specifier %v, adding spaces between arguments and the latter a newline at the end. So `fmt.Print("Hello:", 23)` produces as `output: Hello: 23`