Go Programming Language

Low level programming

Department of Computer Science and Engineering

Go Programming Language

Low level programming

- The design of Go guarantees a number of safety properties that limit the ways in which a Go program can ‘‘go wrong.’’
- During compilation, type checking detects most attempts to apply an operation to a value that is inappropriate for its type, for instance, subtracting one string from another.
- Strict rules for type conversions prevent direct access to the internals of built-in types like strings, maps, slices, and channels.
- For errors that cannot be detected statically, such as out-of-bounds array accesses or nil pointer dereferences, dynamic checks ensure that the program immediately terminates with an informative error whenever a forbidden operation occurs.
- Automatic memory management (garbage collection) eliminates ‘‘use after free’’ bugs, as well as most memory leaks

Go Programming Language

Low level programming

- Many implementation details are inaccessible to Go programs.
- There is no way to discover the memory layout of an aggregate type like a struct, or the machine code for a function, or the identity of the operating system thread on which the current goroutine is running.
- The Go scheduler freely moves goroutines from one thread to another.
- A pointer identifies a variable without revealing the variable's numeric address. Addresses may change as the garbage collector moves variables; pointers are transparently updated
- These features make Go programs, especially failing ones, more predictable and less mysterious than programs in C
- By hiding the underlying details, they also make Go programs highly portable, since the language semantics are largely independent of any particular compiler, operating system, or CPU architecture

Go Programming Language

Unsafe package

- The unsafe package is rather magical.
- Although it appears to be a regular package and is imported in the usual way, it is actually implemented by the compiler.
- It provides access to a number of built-in language features that are not ordinarily available because they expose details of Go's memory layout.
- Presenting these features as a separate package makes the rare occasions on which they are needed more conspicuous.
- Also, some environments may restrict the use of the unsafe package for security reasons.
- Package unsafe is used extensively within low-level packages like runtime, os, syscall, and net that interact with the operating system, but is almost never needed by ordinary programs.

Go Programming Language

unsafe.SizeOf function

- The unsafe.Sizeof function reports the size in bytes of the representation of its operand, which may be an expression of any type; the expression is not evaluated.
- A call to Sizeof is a constant expression of type uintptr, so the result may be used as the dimension of an array type, or to compute other constants.
- Sizeof reports only the size of the fixed part of each data structure, like the pointer and length of a string, but not indirect parts like the contents of the string.
- Typical sizes for all nonaggregate Go types are shown below, though the exact sizes may vary by toolchain.

Type	Size
`bool`	1 byte
`intN`, `uintN`, `floatN`, `complexN`	*N* / 8 bytes (for example, `float64` is 8 bytes)
`int`, `uint`, `uintptr`	1 word
`*T`	1 word
`string`	2 words (data, len)
`[]T`	3 words (data, len, cap)
`map`	1 word
`func`	1 word
`chan`	1 word
`interface`	2 words (type, value)

Go Programming Language

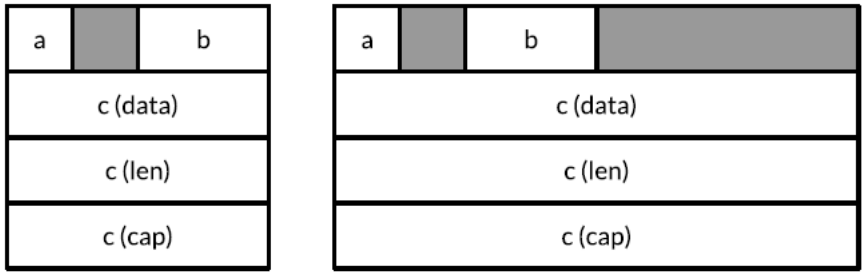
unsafe.SizeOf function

- For portability, we've given the sizes of reference types in terms of words, where a word is 4 bytes on a 32-bit platform and 8 bytes on a 64-bit platform
- Computers load and store values from memory most efficiently when those values are properly aligned. For example, the address of a value of a two-byte type such as int16 should be an even number, the address of a four-byte value such as a rune should be a multiple of four, and the address of an eight-byte value such as a float64, uint64, or 64-bit pointer should be a multiple of eight. Alignment requirements of higher multiples are unusual, even for larger data types such as complex128.
- For this reason, the size of a value of an aggregate type (a struct or array) is at least the sum of the sizes of its fields or elements but may be greater due to the presence of ‘‘holes.’’

Go Programming Language

unsafe.SizeOf function

- Holes are unused spaces added by the compiler to ensure that the following field or element is properly aligned relative to the start of the struct or array.



- The language specification does not guarantee that the order in which fields are declared is the order in which they are laid out in memory, so in theory a compiler is free to rearrange them
- If the types of a struct's fields are of different sizes, it may be more space-efficient to declare the fields in an order that packs them as tightly as possible.

Go Programming Language

unsafe.Alignof function

- The unsafe.Alignof function reports the required alignment of its argument's type.
- Like Sizeof, it may be applied to an expression of any type, and it yields a constant.
- Typically, boolean and numeric types are aligned to their size (up to a maximum of 8 bytes) and all other types are word-aligned.
- The unsafe.Offsetof function, whose operand must be a field selector x.f, computes the offset of field f relative to the start of its enclosing struct x, accounting for holes, if any

```
var x struct {
    a bool
    b int16
    c []int
}
```

```
Typical 32-bit platform:
Sizeof(x)   = 16  Alignof(x)   = 4
Sizeof(x.a) = 1   Alignof(x.a) = 1  Offsetof(x.a) = 0
Sizeof(x.b) = 2   Alignof(x.b) = 2  Offsetof(x.b) = 2
Sizeof(x.c) = 12  Alignof(x.c) = 4  Offsetof(x.c) = 4

Typical 64-bit platform:
Sizeof(x)   = 32  Alignof(x)   = 8
Sizeof(x.a) = 1   Alignof(x.a) = 1  Offsetof(x.a) = 0
Sizeof(x.b) = 2   Alignof(x.b) = 2  Offsetof(x.b) = 2
Sizeof(x.c) = 24  Alignof(x.c) = 8  Offsetof(x.c) = 8
```

Go Programming Language

unsafe.Pointer function

- Most pointer types are written *T, meaning ‘‘a pointer to a variable of type T.’’
- The unsafe.Pointer type is a special kind of pointer that can hold the address of any variable. Of course, we can’t indirect through an unsafe.Pointer using *p because we don’t know what type that expression should have.
- Like ordinary pointers, unsafe.Pointers are comparable and may be compared with nil, which is the zero value of the type.
- An ordinary *T pointer may be converted to an unsafe.Pointer, and an unsafe.Pointer may be converted back to an ordinary pointer, not necessarily of the same type *T.
- By converting a *float64 pointer to a *uint64, for instance, we can inspect the bit pattern of a floating-point variable

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering