builtin

package standard library

Version: go1.20.1 Latest | Published: Feb 14, 2023 | License: BSD-3-Clause | Imports: 0 | Imported by: 0

Details Valid go.mod file Redistributable license Tagged version Stable version

Learn more

Repository cs.opensource.google/go/go

Links Report a Vulnerability

Documentation

Documentation

Overview

Package builtin provides documentation for Go's predeclared identifiers. The items documented here are not actually in package builtin but their descriptions here allow godoc to present documentation for the language's special identifiers.

Index

type ComplexType
type FloatType
type IntegerType
type Type
type Type1
type any
type bool
type byte
type comparable
type complex128
type complex64
type error
type float32
type float64
type int
type int16
type int32
type int64
type int8
type rune
type string
type uint
type uint16
type uint32
type uint64
type uint8
type uintptr

Constants

View Source

```
const (
    true  = 0 == 0 // Untyped bool.
    false = 0 != 0 // Untyped bool.
)
```

true and false are the two untyped boolean values.

View Source

```
const iota = 0 // Untyped int.
```

iota is a predeclared identifier representing the untyped integer ordinal number of the current const specification in a (usually parenthesized) const declaration. It is zero-indexed.

Variables

View Source

```
var nil Type // Type must be a pointer, channel, func, interface, map, or slice type
```

nil is a predeclared identifier representing the zero value for a pointer, channel, func, interface, map, or slice type.

Functions

func append

```
func append(slice []Type, elems ...Type) []Type
```

The append built-in function appends elements to the end of a slice. If it has sufficient capacity, the destination is resliced to accommodate the new elements. If it does not, a new underlying array will be allocated. Append returns the updated slice. It is therefore necessary to store the result of append, often in the variable holding the slice itself:

```
slice = append(slice, elem1, elem2)
slice = append(slice, anotherSlice...)
```

As a special case, it is legal to append a string to a byte slice, like this:

```
slice = append([]byte("hello "), "world"...)
```

func cap

```
func cap(v Type) int
```

The cap built-in function returns the capacity of v, according to its type:

```
Array: the number of elements in v (same as len(v)).
Pointer to array: the number of elements in *v (same as len(v)).
Slice: the maximum length the slice can reach when resliced;
if v is nil, cap(v) is zero.
Channel: the channel buffer capacity, in units of elements;
if v is nil, cap(v) is zero.
```

For some arguments, such as a simple array expression, the result can be a constant. See the Go language specification's "Length and capacity" section for details.

func close

```
func close(c chan<- Type)
```

The close built-in function closes a channel, which must be either bidirectional or send-only. It should be executed only by the sender, never the receiver, and has the effect of shutting down the channel after the last sent value is received. After the last value has been received from a closed channel c, any receive from c will succeed without blocking, returning the zero value for the channel element. The form

```
x, ok := <-c
```

will also set ok to false for a closed and empty channel.

func complex

```
func complex(r, i FloatType) ComplexType
```

The complex built-in function constructs a complex value from two floating-point values. The real and imaginary parts must be of the same size, either float32 or float64 (or assignable to them), and the return value will be the corresponding complex type (complex64 for float32, complex128 for float64).

func copy

```
func copy(dst, src []Type) int
```

The copy built-in function copies elements from a source slice into a destination slice. (As a special case, it also will copy bytes from a string to a slice of bytes.) The source and destination may overlap. Copy returns the number of elements copied, which will be the minimum of len(src) and len(dst).

func delete

```
func delete(m map[Type]Type1, key Type)
```

The delete built-in function deletes the element with the specified key (m[key]) from the map. If m is nil or there is no such element, delete is a no-op.

func imag

```
func imag(c ComplexType) FloatType
```

The imag built-in function returns the imaginary part of the complex number c. The return value will be floating point type corresponding to the type of c.

func len

```
func len(v Type) int
```

The len built-in function returns the length of v, according to its type:

```
Array: the number of elements in v.
Pointer to array: the number of elements in *v (even if v is nil).
Slice, or map: the number of elements in v; if v is nil, len(v) is zero.
String: the number of bytes in v.
Channel: the number of elements queued (unread) in the channel buffer;
         if v is nil, len(v) is zero.
```

For some arguments, such as a string literal or a simple array expression, the result can be a constant. See the Go language specification's "Length and capacity" section for details.

func make

```
func make(t Type, size ...IntegerType) Type
```

The make built-in function allocates and initializes an object of type slice, map, or chan (only). Like new, the first argument is a type, not a value. Unlike new, make's return type is the same as the type of its argument, not a pointer to it. The specification of the result depends on the type:

```
Slice: The size specifies the length. The capacity of the slice is
equal to its length. A second integer argument may be provided to
specify a different capacity; it must be no smaller than the
length. For example, make([]int, 0, 10) allocates an underlying array
of size 10 and returns a slice of length 0 and capacity 10 that is
backed by this underlying array.
Map: An empty map is allocated with enough space to hold the
specified number of elements. The size may be omitted, in which case
a small starting size is allocated.
Channel: The channel's buffer is initialized with the specified
buffer capacity. If zero, or the size is omitted, the channel is
unbuffered.
```

func new

```
func new(Type) *Type
```

The new built-in function allocates memory. The first argument is a type, not a value, and the value returned is a pointer to a newly allocated zero value of that type.

func panic

```
func panic(v any)
```

The panic built-in function stops normal execution of the current goroutine. When a function F calls panic, normal execution of F stops immediately. Any functions whose execution was deferred by F are run in the usual way, and then F returns to its caller. To the caller G, the invocation of F then behaves like a call to panic, terminating G's execution and running any deferred functions. This continues until all functions in the executing goroutine have stopped, in reverse order. At that point, the program is terminated with a non-zero exit code. This termination sequence is called panicking and can be controlled by the built-in function recover.

func print

added in go1.2

```
func print(args ...Type)
```

The print built-in function formats its arguments in an implementation-specific way and writes the result to standard error. Print is useful for bootstrapping and debugging; it is not guaranteed to stay in the language.

func println

added in go1.2

```
func println(args ...Type)
```

The println built-in function formats its arguments in an implementation-specific way and writes the result to standard error. Spaces are always added between arguments and a newline is appended. Println is useful for bootstrapping and debugging; it is not guaranteed to stay in the language.

func real

```
func real(c ComplexType) FloatType
```

The real built-in function returns the real part of the complex number c. The return value will be floating point type corresponding to the type of c.

func recover

```
func recover() any
```

The recover built-in function allows a program to manage behavior of a panicking goroutine. Executing a call to recover inside a deferred function (but not any function called by it) stops the panicking sequence by restoring normal execution and retrieves the error value passed to the call of panic. If recover is called outside the deferred function it will not stop a panicking sequence. In this case, or when the goroutine is not panicking, or if the argument supplied to panic was nil, recover returns nil. Thus the return value from recover reports whether the goroutine is panicking.

Types

type ComplexType

```
type ComplexType complex64
```

ComplexType is here for the purposes of documentation only. It is a stand-in for either complex type: complex64 or complex128.

type FloatType

```
type FloatType float32
```

FloatType is here for the purposes of documentation only. It is a stand-in for either float type: float32 or float64.

type IntegerType

```
type IntegerType int
```

IntegerType is here for the purposes of documentation only. It is a stand-in for any integer type: int, uint, int8 etc.

type Type

```
type Type int
```

Type is here for the purposes of documentation only. It is a stand-in for any Go type, but represents the same type for any given function invocation.

type Type1

```
type Type1 int
```

Type1 is here for the purposes of documentation only. It is a stand-in for any Go type, but represents the same type for any given function invocation.

type any

added in go1.18

```
type any = interface{}
```

any is an alias for interface{} and is equivalent to interface{} in all ways.

type bool

```
type bool bool
```

bool is the set of boolean values, true and false.

type byte

```
type byte = uint8
```

byte is an alias for uint8 and is equivalent to uint8 in all ways. It is used, by convention, to distinguish byte values from 8-bit unsigned integer values.

type comparable

added in go1.18

```
type comparable interface{ comparable }
```

comparable is an interface that is implemented by all comparable types (booleans, numbers, strings, pointers, channels, arrays of comparable types, structs whose fields are all comparable types). The comparable interface may only be used as a type parameter constraint, not as the type of a variable.

type complex128

```
type complex128 complex128
```

complex128 is the set of all complex numbers with float64 real and imaginary parts.

type complex64

```
type complex64 complex64
```

complex64 is the set of all complex numbers with float32 real and imaginary parts.

type error

```
type error interface {
    Error() string
}
```

The error built-in interface type is the conventional interface for representing an error condition, with the nil value representing no error.

type float32

```
type float32 float32
```

float32 is the set of all IEEE-754 32-bit floating-point numbers.

type float64

```
type float64 float64
```

float64 is the set of all IEEE-754 64-bit floating-point numbers.

type int

```
type int int
```

int is a signed integer type that is at least 32 bits in size. It is a distinct type, however, and not an alias for, say, int32.

type int16

```
type int16 int16
```

int16 is the set of all signed 16-bit integers. Range: -32768 through 32767.

type int32

```
type int32 int32
```

int32 is the set of all signed 32-bit integers. Range: -2147483648 through 2147483647.

type int64

```
type int64 int64
```

int64 is the set of all signed 64-bit integers. Range: -9223372036854775808 through 9223372036854775807.

type int8

```
type int8 int8
```

int8 is the set of all signed 8-bit integers. Range: -128 through 127.

type rune

```
type rune = int32
```

rune is an alias for int32 and is equivalent to int32 in all ways. It is used, by convention, to distinguish character values from integer values.

type string

```
type string string
```

string is the set of all strings of 8-bit bytes, conventionally but not necessarily representing UTF-8-encoded text. A string may be empty, but not nil. Values of string type are immutable.

type uint

```
type uint uint
```

uint is an unsigned integer type that is at least 32 bits in size. It is a distinct type, however, and not an alias for, say, uint32.

type uint16

```
type uint16 uint16
```

uint16 is the set of all unsigned 16-bit integers. Range: 0 through 65535.

type uint32

```
type uint32 uint32
```

uint32 is the set of all unsigned 32-bit integers. Range: 0 through 4294967295.

type uint64

```
type uint64 uint64
```

uint64 is the set of all unsigned 64-bit integers. Range: 0 through 18446744073709551615.

type uint8

```
type uint8 uint8
```

uint8 is the set of all unsigned 8-bit integers. Range: 0 through 255.

type uintptr

```
type uintptr uintptr
```

uintptr is an integer type that is large enough to hold the bit pattern of any pointer.

Source Files

View all

builtin.go