Package unsafe

```
import "unsafe"
```


Overview ▾

Package unsafe contains operations that step around the type safety of Go programs.

Packages that import unsafe may be non-portable and are not protected by the Go 1 compatibility guidelines.

Index ▾

Package files

unsafe.go

func Alignof

```
func Alignof(x ArbitraryType) uintptr
```

Alignof takes an expression x of any type and returns the required alignment of a hypothetical variable v as if v was declared via var v = x. It is the largest value m such that the address of v is always zero mod m. It is the same as the value returned by reflect.TypeOf(x).Align(). As a special case, if a variable s is of struct type and f is a field within that struct, then Alignof(s.f) will return the required alignment of a field of that type within a struct. This case is the same as the value returned by reflect.TypeOf(s.f).FieldAlign(). The return value of Alignof is a Go constant.

func Offsetof

```
func Offsetof(x ArbitraryType) uintptr
```

Offsetof returns the offset within the struct of the field represented by x, which must be of the form structValue.field. In other words, it returns the number of bytes between the start of the struct and the start of the field. The return value of Offsetof is a Go constant.

func Sizeof

```
func Sizeof(x ArbitraryType) uintptr
```

Sizeof takes an expression x of any type and returns the size in bytes of a hypothetical variable v as if v was declared via var v = x. The size does not include any memory possibly referenced by x. For instance, if x is a slice, Sizeof returns the size of the slice descriptor, not the size of the memory referenced by the slice. The return value of Sizeof is a Go constant.

type ArbitraryType

ArbitraryType is here for the purposes of documentation only and is not actually part of the unsafe package. It represents the type of an arbitrary Go expression.

```
type ArbitraryType int
```

type Pointer

Pointer represents a pointer to an arbitrary type. There are four special operations available for type Pointer that are not available for other types:

```
- A pointer value of any type can be converted to a Pointer.
- A Pointer can be converted to a pointer value of any type.
- A uintptr can be converted to a Pointer.
- A Pointer can be converted to a uintptr.
```

Pointer therefore allows a program to defeat the type system and read and write arbitrary memory. It should be used with extreme care.

The following patterns involving Pointer are valid. Code not using these patterns is likely to be invalid today or to become invalid in the future. Even the valid patterns below come with

important caveats.

Running "go vet" can help find uses of Pointer that do not conform to these patterns, but silence from "go vet" is not a guarantee that the code is valid.

(1) Conversion of a *T1 to Pointer to *T2.

Provided that T2 is no larger than T1 and that the two share an equivalent memory layout, this conversion allows reinterpreting data of one type as data of another type. An example is the implementation of math.Float64bits:

```
func Float64bits(f float64) uint64 {
        return *(*uint64)(unsafe.Pointer(&f))
}
```

(2) Conversion of a Pointer to a uintptr (but not back to Pointer).

Converting a Pointer to a uintptr produces the memory address of the value pointed at, as an integer. The usual use for such a uintptr is to print it.

Conversion of a uintptr back to Pointer is not valid in general.

A uintptr is an integer, not a reference. Converting a Pointer to a uintptr creates an integer value with no pointer semantics. Even if a uintptr holds the address of some object, the garbage collector will not update that uintptr's value if the object moves, nor will that uintptr keep the object from being reclaimed.

The remaining patterns enumerate the only valid conversions from uintptr to Pointer.

(3) Conversion of a Pointer to a uintptr and back, with arithmetic.

If p points into an allocated object, it can be advanced through the object by conversion to uintptr, addition of an offset, and conversion back to Pointer.

```
p = unsafe.Pointer(uintptr(p) + offset)
```

The most common use of this pattern is to access fields in a struct or elements of an array:

```
// equivalent to f := unsafe.Pointer(&s.f)
f := unsafe.Pointer(uintptr(unsafe.Pointer(&s)) + unsafe.Offsetof(s.f))

// equivalent to e := unsafe.Pointer(&x[i])
e := unsafe.Pointer(uintptr(unsafe.Pointer(&x[0])) +
i*unsafe.Sizeof(x[0]))
```

It is valid both to add and to subtract offsets from a pointer in this way. It is also valid to use &^ to round pointers, usually for alignment. In all cases, the result must continue to point into the original allocated object.

Unlike in C, it is not valid to advance a pointer just beyond the end of its original allocation:

```
// INVALID: end points outside allocated space.
var s thing
end = unsafe.Pointer(uintptr(unsafe.Pointer(&s)) + unsafe.Sizeof(s))

// INVALID: end points outside allocated space.
b := make([]byte, n)
end = unsafe.Pointer(uintptr(unsafe.Pointer(&b[0])) + uintptr(n))
```

Note that both conversions must appear in the same expression, with only the intervening arithmetic between them:

```
// INVALID: uintptr cannot be stored in variable
// before conversion back to Pointer.
u := uintptr(p)
p = unsafe.Pointer(u + offset)
```

Note that the pointer must point into an allocated object, so it may not be nil.

```
// INVALID: conversion of nil pointer
u := unsafe.Pointer(nil)
p := unsafe.Pointer(uintptr(u) + offset)
```

(4) Conversion of a Pointer to a uintptr when calling syscall.Syscall.

The Syscall functions in package syscall pass their uintptr arguments directly to the operating system, which then may, depending on the details of the call, reinterpret some of them as pointers. That is, the system call implementation is implicitly converting certain arguments back from uintptr to pointer.

If a pointer argument must be converted to uintptr for use as an argument, that conversion must appear in the call expression itself:

```
syscall.Syscall(SYS_READ, uintptr(fd), uintptr(unsafe.Pointer(p)),
uintptr(n))
```

The compiler handles a Pointer converted to a uintptr in the argument list of a call to a function implemented in assembly by arranging that the referenced allocated object, if any, is retained and not moved until the call completes, even though from the types alone it would appear that the object is no longer needed during the call.

For the compiler to recognize this pattern, the conversion must appear in the argument list:

```
// INVALID: uintptr cannot be stored in variable
// before implicit conversion back to Pointer during system call.
u := uintptr(unsafe.Pointer(p))
syscall.Syscall(SYS_READ, uintptr(fd), u, uintptr(n))
```

(5) Conversion of the result of reflect.Value.Pointer or reflect.Value.UnsafeAddr from uintptr to Pointer.

Package reflect's Value methods named Pointer and UnsafeAddr return type uintptr instead of unsafe.Pointer to keep callers from changing the result to an arbitrary type without first importing "unsafe". However, this means that the result is fragile and must be converted to Pointer immediately after making the call, in the same expression:

```
p := (*int)(unsafe.Pointer(reflect.ValueOf(new(int)).Pointer()))
```

As in the cases above, it is invalid to store the result before the conversion:

```
// INVALID: uintptr cannot be stored in variable
// before conversion back to Pointer.
u := reflect.ValueOf(new(int)).Pointer()
p := (*int)(unsafe.Pointer(u))
```

(6) Conversion of a reflect.SliceHeader or reflect.StringHeader Data field to or from Pointer.

As in the previous case, the reflect data structures SliceHeader and StringHeader declare the field Data as a uintptr to keep callers from changing the result to an arbitrary type without first importing "unsafe". However, this means that SliceHeader and StringHeader are only valid when interpreting the content of an actual slice or string value.

```
var s string
hdr := (*reflect.StringHeader)(unsafe.Pointer(&s)) // case 1
hdr.Data = uintptr(unsafe.Pointer(p))              // case 6 (this case)
hdr.Len = n
```

In this usage hdr.Data is really an alternate way to refer to the underlying pointer in the string header, not a uintptr variable itself.

In general, reflect.SliceHeader and reflect.StringHeader should be used only as *reflect.SliceHeader and *reflect.StringHeader pointing at actual slices or strings, never as plain structs. A program should not declare or allocate variables of these struct types.

```
// INVALID: a directly-declared header will not hold Data as a reference.
var hdr reflect.StringHeader
hdr.Data = uintptr(unsafe.Pointer(p))
hdr.Len = n
s := *(*string)(unsafe.Pointer(&hdr)) // p possibly already lost
```

```
type Pointer *ArbitraryType
```