

Pointers

01

Memory

02

Creation & Usage

03

Visualization

Memory

- | Function calls in Go are "pass by value"
 - | A copy of each function argument is made, regardless of size
 - | Potentially slow for large data structures
 - | More difficult to manage program state
- | This can be changed by using **pointers**

Pointers

- | Pointers are variables that "point to" memory
- | The value of the variable itself is a memory address
 - | Accessing the data requires **dereferencing** the pointer
 - | This allows changing values that exist elsewhere in the program

Creating Pointers

- | Asterisk (*) when used with a type indicates the value is a pointer
- | Ampersand (&) creates a pointer from a variable

```
value := 10
```

```
var valuePtr *int  
valuePtr = &value
```

```
value := 10
```

```
valuePtr := &value
```

Using Pointers

- | Asterisk (*) when used with a pointer will dereference the pointer
 - | This provides access to the actual data it points to

```
func increment(x *int) {  
    *x += 1  
}  
  
i := 1  
increment(&i)  
// i == 2
```

Pointers Visualized

Address	Data				
0x07A	<table border="1"><tr><td>9</td><td>9</td><td>8</td></tr></table>	9	9	8	big := 998
9	9	8			
0x07D	<table border="1"><tr><td>0</td><td>7</td><td>A</td></tr></table>	0	7	A	bigPtr := &big
0	7	A			
0x07A	<table border="1"><tr><td>9</td><td>9</td><td>9</td></tr></table>	9	9	9	*bigPtr += 1 // big == 999
9	9	9			

Recap

- | Pointers are used to modify data that exists outside of a function
- | Asterisk (*) on a type indicates the type is a pointer
- | Ampersand (&) creates a pointer
- | Asterisk (*) on a variable will **dereference** the pointer
 - | Operations on a dereferenced pointer occur on the original data