Basic Pointer Operations Go

```go
package main
import "fmt"

func main() {
    x := 10
    p := &x
    fmt.Println(*p)
    *p = 20
    fmt.Println(x)
}
```

Nil Pointer Go

```go
package main
import "fmt"

func main() {
    var p *int
    fmt.Println(p)
    fmt.Println(p == nil)
}
```

Nil Pointer Go

```go
package main
import "fmt"

func main() {
    var p *int
    fmt.Println(*p) // What happens?
}
```

Pointer to Pointer Go

```go
package main
import "fmt"

func main() {
    x := 10
    p := &x
    pp := &p
    fmt.Println(**pp)
    **pp = 20
    fmt.Println(x)
}
```

New Function Go

```go
package main
import "fmt"

func main() {
    p := new(int)
    fmt.Println(*p)
    *p = 42
    fmt.Println(*p)
}
```

Passing by Value vs Pointer Go

```go
package main
import "fmt"

func modifyValue(x int) {
    x = 20
}

func modifyPointer(p *int) {
    *p = 20
}

func main() {
    a := 10
    modifyValue(a)
    fmt.Println(a)

    b := 10
    modifyPointer(&b)
    fmt.Println(b)
}
```

Returning Pointer to Local Variable Go

```go
package main
import "fmt"

func makePointer() *int {
    x := 10
    return &x
}

func main() {
    p := makePointer()
    fmt.Println(*p)
}
```

Struct Pointer Shorthand Go

```go
package main
import "fmt"

type Person struct {
    name string
    age  int
}

func main() {
    p := &Person{name: "Alice", age: 30}
    fmt.Println(p.name)
    fmt.Println((*p).name)
}
```

Slice Pointer Confusion Go

```go
package main
import "fmt"

func modify(s []int) {
    s[0] = 100
}

func main() {
    nums := []int{1, 2, 3}
    modify(nums)
    fmt.Println(nums)
}
// Does this modify the original slice?
```

When NOT to Use Pointers Go

```go
package main
import "fmt"

func addPointer(a, b *int) int {
    return *a + *b
}

func addValue(a, b int) int {
    return a + b
}

func main() {
    x, y := 5, 10
    fmt.Println(addValue(x, y))
}
// Which is better?
```

Modifying Slice Header Go

```go
package main
import "fmt"

func appendValue(s []int) {
    s = append(s, 4)
}

func appendPointer(s *[]int) {
    *s = append(*s, 4)
}

func main() {
    nums1 := []int{1, 2, 3}
    appendValue(nums1)
    fmt.Println(nums1)

    nums2 := []int{1, 2, 3}
    appendPointer(&nums2)
    fmt.Println(nums2)
}

// Which modifies the original?

```

Map Pointer Confusion Go

```go
package main
import "fmt"

func modify(m map[string]int) {
    m["a"] = 100
}

func main() {
    myMap := map[string]int{"a": 1}
    modify(myMap)
    fmt.Println(myMap)
}
// Does this modify the original map?
```

Nil Map vs Empty Map — Go

```go
package main
import "fmt"

func main() {
    var m1 map[string]int  // nil map
    m2 := map[string]int{} // empty map

    fmt.Println(m1 == nil)
    fmt.Println(m2 == nil)

    // m1["a"] = 1 // What happens?
    m2["a"] = 1
    fmt.Println(m2)
}
```

Zero Value vs Nil for Optional Values Go

```go
package main
import "fmt"

type Config struct {
    maxSize int
    timeout *int
}

func main() {
    c1 := Config{maxSize: 100}
    c2 := Config{maxSize: 100, timeout: new(int)}

    fmt.Println(c1.timeout)
    fmt.Println(c2.timeout == nil)
}
// How to distinguish "not set" from "set to zero"?
```

Struct with Pointer Fields Go

```go
package main
import "fmt"

type Node struct {
    value int
    next  *Node
}

func main() {
    n1 := Node{value: 1}
    n2 := Node{value: 2}
    n1.next = &n2

    fmt.Println(n1.next.value)
}

//What's this pattern?
```

Comparing Pointers Go

```go
package main
import "fmt"

func main() {
    a := 10
    b := 10

    p1 := &a
    p2 := &a
    p3 := &b

    fmt.Println(p1 == p2)
    fmt.Println(p1 == p3)
    fmt.Println(*p1 == *p3)
}
// What do the comparisons check?
```

Comparing Pointers Go

```go
package main
import "fmt"

type Counter struct {
    count int
}

func (c *Counter) Increment() {
    c.count++
}

func (c Counter) Value() int {
    return c.count
}

func main() {
    c := Counter{}
    c.Increment()
    c.Increment()
    fmt.Println(c.Value())
}

// Which receiver type should we use?
```

Large Struct Performance Go

```go
package main
import "fmt"

type LargeStruct struct {
    data [1000]int
}


func processValue(s LargeStruct) {
    fmt.Println(s.data[0])
}


func processPointer(s *LargeStruct) {
    fmt.Println(s.data[0])
}

func main() {
    large := LargeStruct{}
    processValue(&large)
}
// When should we use pointer?
```

Slice of Pointers vs Slice of Values Go

```go
package main
import "fmt"

type Person struct {
    name string
}

func main() {
    // Slice of values
    people1 := []Person{
        {name: "Alice"},
        {name: "Bob"},
    }
    people1[0].name = "Charlie"
    fmt.Println(people1[0].name)

    // Slice of pointers
    people2 := []*Person{
        {name: "Alice"},
        {name: "Bob"},
    }
    people2[0].name = "Charlie"
    fmt.Println(people2[0].name)
}
// When to use slice of pointers?
```

Pointer Receiver and Value Receiver Mix Go

```go
package main
import "fmt"

type Counter struct {
    count int
}

func (c *Counter) Increment() {
    c.count++
}

func (c Counter) GetCount() int {
    return c.count
}

func main() {
    c := &Counter{}
    c.Increment()
    fmt.Println(c.GetCount())
}

// Can pointer call value receiver method?
```

Escape Analysis Example Go

```go
package main

func createOnStack() int {
    x := 10
    return x
}

func createOnHeap() *int {
    x := 10
    return &x
}

func main() {
    a := createOnStack()
    b := createOnHeap()
    println(a, *b)
}

// Where are variables allocated?
```

Pointer Zero Value Initialization Go

```go
package main
import "fmt"

type Config struct {
    Name    string
    Timeout *int
}

func main() {
    c := Config{Name: "app"}

    if c.Timeout != nil {
        fmt.Println("Timeout:", *c.Timeout)
    } else {
        fmt.Println("No timeout set")
    }
}

// Pattern for optional config?
```

Slices Share Underlying Array Go

```go
package main
import "fmt"

func modify(s []int) {
    s[0] = 999
    s = append(s, 4)
    s[1] = 888
}

func main() {
    nums := []int{1, 2, 3}
    modify(nums)
    fmt.Println(nums)
}

// What's in nums after modify?
```

Map Value Not Addressable Go

```go
package main
import "fmt"

type Person struct {
    name string
    age  int
}

func main() {
    people := map[string]Person{
        "alice": {name: "Alice", age: 30},
    }

    // This won't compile:
    // people["alice"].age = 31

    // Must do this:
    p := people["alice"]
    p.age = 31
    people["alice"] = p

    fmt.Println(people["alice"].age)
}
// Why can't we modify map value directly?
```

Function Pointer Parameter Modification Go

```go
package main
import "fmt"

func swap(a, b *int) {
    temp := *a
    *a = *b
    *b = temp
}

func main() {
    x, y := 10, 20
    fmt.Println(x, y)
    swap(&x, &y)
    fmt.Println(x, y)
}
// What happens to x and y?
```

🎯 BONUS CHALLENGE Go

```go
package main
import "fmt"

type Data struct {
    value int
}

func modify1(d Data) Data {
    d.value = 100
    return d
}

func modify2(d *Data) {
    d.value = 200
}

func modify3(d *Data) *Data {
    d.value = 300
    return d
}

func main() {
    d := Data{value: 1}

    d = modify1(d)
    fmt.Println(d.value)

    modify2(&d)
    fmt.Println(d.value)

    d2 := modify3(&d)
    fmt.Println(d.value)
    fmt.Println(d2.value)
    fmt.Println(&d == d2)
}
```

🎯 BONUS CHALLENGE Markdown

```
**Pointer Basics:**
- `&` gets address (reference)
- `*` dereferences (reads/writes value)
- Zero value is `nil`
- Dereferencing nil = panic

**When to Use Pointers:**
- ✅ To modify function parameters
- ✅ For large structs (avoid copying)
- ✅ For optional values (nil = not set)
- ✅ For recursive data structures
- ❌ For small types (int, bool)
- ❌ For slices/maps (already references)

**Pointer Receivers:**
- Use `*T` to modify struct
- Use `T` for read-only methods
- Go auto-converts between them

**Reference Types:**
- Slices are references (pointer + len + cap)
- Maps are references
- Don't need pointer to slice/map for modifications
- Exception: modifying slice header needs pointer

**Nil Semantics:**
- Reading nil map: OK (returns zero)
- Writing nil map: PANIC
- Dereferencing nil pointer: PANIC
- Use nil for "not provided" pattern

**Performance:**
- Small types: pass by value (faster)
- Large structs: pass by pointer
- Compiler does escape analysis
- Stack allocation faster than heap

**Common Patterns:**
- Optional config: `*int` (nil = not set)
- Linked structures: `*Node`
- Modify in place: pointer parameter
- Map of pointers: `map[K]*V` for modifications

**Pitfalls:**
- Don't use `*[]T` or `*map[K]V` (redundant)
- Map values not addressable
- Slice modifications after append unreliable
- Always check nil before dereference

**Idiomatic Go:**
- Prefer values when possible
- Pointers are for mutation or size
- "Don't Fear the Pointers" but don't overuse
- Pointers are last resort, not first choice
```