Chapter 14 포인터

날짜	@November 17, 2021
발표자	김선영

14.1 포인터란?

포인터는 메모리 주소를 값으로 갖는 타입. 예) int 타입 변수 a. a는 메모리에 저장되어있고 속성으로 메모리 주소를 가지고 있음. 변수 a의 주소가 0x0100번지라면 메모리 주솟값 또한 숫자 값이기 때문에 다른 변수의 값으로 사용될 수 있다 → 포인터 변수

```
p = &a
```

포인터 변수 p에 a의 주소를 대입하는 구문. 포인터 변수 p의 값은 변수 a의 주소인 0x0100. '포인터 변수 p가 변수 a를 가리킨다'고 표현.

포인터를 이용하면 여러 포인터 변수가 하나의 메모리 공간을 가리킬 수도 있고 포인터가 가리키고 있는 메모리의 값을 읽을 수도 변경할 수도 있다.

14.1.1 포인터 변수 선언

포인터 변수는 가리키는 데이터 타입 앞에 *을 붙여 선언

```
var p *int
```

p는 int타입 데이터의 메모리 주소를 가리키는 포인터 변수. float64 → *float64, User 구조체 → *User

```
var a int
var p *int
p = &a
```

변수 a의 메모리 주소를 포인터 변수 p의 값으로 대입. p를 이용해서 변수 a의 값을 변경할 수 있다. 포인터 변수 앞에 *를 붙여 그 포인터 변수가 가리키는 메모리 공간에 접근 가능

```
*p = 20
```

a의 값이 20으로 변경되었음.

```
package main

import "fmt"

func main() {
  var a int = 500
  var p *int // int 포인터 변수 p 선언

  p = &a // a의 메모리 주소를 변수 p의 값으로 대입

  fmt.Printf("p의 값: %p\n", p)             // 메모리 주솟값 출력
  fmt.Printf("p가 가리키는 메모리의 값: %d\n", *p) // p가 가리키는 메모리의 값 출력
  *p = 100                                 // p가 가리키는 메모리 공간의 값을 변경합니다.
  fmt.Printf("a의 값: %d\n", a)             // a값 변화 확인
}
```

출력문

```
p의 값: 0xc0000140a0
p가 가리키는 메모리의 값: 500
a의 값: 100
```

- 메모리 주솟값은 %p로 출력

14.1.2 포인터 변숫값 비교하기

== 연산을 사용해 포인터가 같은 메모리 공간을 가리키는지 확인 가능

```
package main

import "fmt"

func main() {
  var a int = 10
  var b int = 20
```

```
  var p1 *int = &a
  var p2 *int = &a
  var p3 *int = &b

  fmt.Printf("p1 == p2 : %v\n", p1 == p2)
  fmt.Printf("p2 == p3 : %v\n", p2 == p3)
}
```

출력문

```
p1 == p2 : true
p2 == p3 : false
```

14.1.3 포인터의 기본값 nil

포인터 변숫값을 초기화하지 않으면 기본값은 nil. 이 값은 0이지만 정확한 의미는 유효하지 않은 메모리 주솟값 즉 어떤 메모리 공간도 가리키고 있지 않음을 나타낸다.

```
var p *int
if p != nul {
  // p가 nil이 아니라는 얘기는 p가 유효한 메모리 주소를 가리킨다는 뜻
}
```

14.2 포인터는 왜 쓰나?

변수 대입이나 함수 인수 전달은 항상 값을 복사하기 때문에 많은 메모리 공간을 사용하는 문제와 큰 메모리 공간을 복사할 때 발생하는 성능 문제를 안고 있다. 또한 다른 공간으로 복사되기 때문에 변경 사항이 적용되지도 않는다.

포인터를 사용하지 않는 예

```
package main

import "fmt"

type Data struct { //Data형 구조체
  value int
  data  [200]int
}

func ChangeData(arg Data) {
  arg.value = 999
  arg.data[100] = 999
```

```
}

func main() {
  var data Data

  ChangeData(data)
  fmt.Printf("value = %d\n", data.value)
  fmt.Printf("data[100] = %d\n", data.data[100])
}
```

출력문

```
value = 0
data[100] = 0
```

- ChangeDate() 함수는 Data 타입 구조체를 매개변수로 받음. 매개변수 arg와 data는 서로 다른 메모리 공간을 갖는 변수.
- arg 매개변숫값을 변경해도 data 변수와 다른 메모리 공간을 가지기 때문에 data 값은 변경되지 않는다.
- ChangeData() 함수 한 번 호출할 때마다 1608 바이트가 복사된다. 만약 짧은 시간에 많이 호출되면 성능 문제 발생

포인터를 이용한 예

```
package main

import "fmt"

type Data struct {
  value int
  data  [200]int
}

func ChangeData(arg *Data) {
  arg.value = 999
  arg.data[100] = 999
}

func main() {
  var data Data

  ChangeData(&data)
  fmt.Printf("value = %d\n", data.value)
```

```
  fmt.Printf("data[100] = %d\n", data.data[100])
}
```

출력문

```
value = 999
data[100] = 999
```

- data 변숫값이 아니라 data의 메모리 주소를 인수로 전달해 8 바이트만 복사. arg 포인터 변수가 가리키는 구조체의 값을 변경.
- 포인터를 이용하면 더 효율적으로 데이터 조작 가능

14.2.1 Data 구조체를 생성해 포인터 변수 초기화하기

구조체 변수를 별도로 생성하지 않고 곧바로 포인터 변수에 구조체를 생성해 주소를 초깃값으로 대입하는 방법

기존 방식

```
var data Data
var p *Data = &data
```

구조체를 생성해 초기화하는 방식

```
var p *Data = &Data{}
```

Data 타입 포인터 변수 p에 Data 구조체를 생성해 그 주소를 대입한다. 이렇게 하면 (메모리에 실제로 있는 구조체 데이터의 실체를 가리키게 되므로) 포인터 변수 p만 가지고도 구조체의 필드값에 접근하고 변경할 수 있다.

14.3 인스턴스

인스턴스란 메모리에 할당된 데이터의 실체를 말한다.

```
var data Data
var p *Data = &data
```

data는 할당된 메모리 공간의 실체인 인스턴스. 포인터 변수 p는 data를 가리킨다. p가 생성될 때 새로운 Data 인스턴스가 만들어진 게 아니라 기존의 data 인스턴스를 가리킨다. 즉, 만들어진 총 Data 인스턴스 개수는 한 개이다.

인스턴스를 별도로 생성하지 않고, 곧바로 인스턴스 생성해 그 주소를 포인터 변수에 초깃값으로 대입하는 코드

```
var p *Data = &Data{}
```

포인터 변수가 아무리 많아도 인스턴스가 추가로 생성되는 것은 아니다.

```
var p1 *Data = &Data{}
var p2 *Data = p1
var p3 *Data = p1
```

한 인스턴스를 포인터 변수 p1, p2, p3가 가리킨다. 가리키는 포인터 변수 개수는 인스턴스 개수와 무관하다.

14.3.1 인스턴스는 데이터의 실체다

인스턴스는 메모리에 존재하는 데이터의 실체이다. 포인터를 이용해서 인스턴스에 접근 가능. 구조체 포인터를 함수 매개변수로 받는다는 말은 인스턴스로 입력을 받겠다는 얘기.

14.3.2 new() 내장 함수

앞서 포인터값을 별도의 변수를 선언하지 않고 초기화하는 방법을 봤다. new 내장 함수를 이용하면 더 간단히 표현 가능하다.

```
p1 := &Data{}
var p2 = new(Data)
```

new() 내장 함수는 인스로 타입을 받는다. 타입을 메모리에 할당하고 기본값으로 채워 그 주소를 반환한다. new를 이용해서 내부 필드값을 원하는 값으로 초기화할수는 없다. &를 사용하면 초기화 가능하다.

14.3.3 인스턴스는 언제 사라지나

만약 메모리에 데이터가 할당만 되고 사라지지 않는다면 프로그램은 금세 메모리가 고갈되어 프로그램이 비정상 종료될 것이다. 그래서 쓸모없는 데이터를 메모리에서 해제하는 기능이 필요하다. Go 언어는 가비지 컬렉터라는 메모리 청소부 기능을 제공한다.

사용되는 데이터인지 아닌지? '아무도 찾지 않는 데이터는 쓸모없는 데이터이다.'

```
func TestFunc(){
  u := &User{}
  u.Age = 30
  fmt.Println(u)
}
```

u 포인터 변수 선언, 인스턴스 생성 → 메모리에 User 데이터 할당 & u 포인터 변수가 가리킴. 이 인스턴스는 u 포인터 변수로 사용되는 인스턴스이기 때문에 지워지면 안된다.

하지만 TestFunc() 종료되면 함수 내부 변수 u는 사라져 User 인스턴스를 가리키는 포인터 변수가 없어 인스턴스는 쓸모가 없게 된다. 가비지 컬렉터는 다음번 청소를 할 때 이 User 인스턴스를 지우게 된다.

메모리는 굉장히 크기 때문에 모두 검사해서 쓸모없는 데이터를 지워주는 데 성능을 많이 쓴다. 가비지 컬렉터를 사용하면 메모리 관리에서 이득을 보지만 성능에서 손해가 발생한다.

14.4 스택 메모리와 힙 메모리

대부분 프로그래밍 언어는 메모리를 할당할 때 스택 메모리 영역 또는 힙 메모리 영역을 사용한다. 이론상 스택이 힙보다 훨씬 효율적이기 때문에 스택에서 메모리를 할당하는 게 더 좋지만 스택 메모리는 함수 내부에만 사용 가능한 영역이다. 그래서 함수 외부로 공개되는 메모리 공가는 힙에서 할당한다. C/C++언어는 malloc() 함수를 직접 호출해서 힙 메모리 공간을 할당하고 자바는 클래스 타입을 힙에, 기본 타입을 스택에 할당한다. Go언어는 탈출 검사를 해서 어느 메모리에 할당할지 결정한다.

함수 외부로 공개되는 인스턴스의 경우 함수가 종료되어도 사라지지 않는다.

```
package main

import "fmt"

type User struct {
  Name string
  Age  int
}

func NewUser(name string, age int) *User {
  var u = User{name, age}
  return &u // 탈출 분석으로 u 메모리가 사라지지 않음
}

func main() {
  userPointer := NewUser("AAA", 23)
```

```
  fmt.Println(userPointer)
}
```

출력문

```
&{AAA 23}
```

- NewUser() 함수에서 선언한 u변수를 반환한다. 함수 내부에서 선언된 변수는 함수가 종료되면 사라진다. 이 코드는 이미 사라진 메모리를 가리키는 댕글링 오류가 발생해야 한다.
- 잘 동작하는 이유? Go언어에서는 탈출 검사를 통해서 u 변수의 인스턴스가 함수 외부로 공개되는 것을 분석해내서 u를 스택 메모리가 아닌 힙 메모리에서 할당한다. 즉 Go언어는 어떤 타입이나 메모리 할당에 의해서 스택을 사용할지 힙을 사용할지 결정하는 게 아니라 **메모리 공간이 함수 외부로 공개되는지 여부를 자동으로 검사해서 스택에 할당할지 힙에 할당할지 결정한다.**
- Go 언어에서 스택 메모리는 계속 증가되는 동적 메모리 풀이다. 일정한 크기를 갖는 C/C++언어와 비교해 메모리 효율성이 높고, 재귀 호출 때문에 스택 메모리가 고갈되는 문제도 발생하지 않는다.

연습문제

1. 8

2.

```
package main

import "fmt"

type Actor struct {
  Name  string
  HP    int
  Speed float64
}

func NewActor(name string, hp int, speed float64) *Actor {
  return &Actor{name, hp, speed}
}
func main() {
  var actor = NewActor("금토끼", 99, 100)
```

```
  fmt.Println(actor.Speed)
  fmt.Println(actor.Name)
}
```

3. 1개 (u 인스턴스)