



# Chapter 13

📅 날짜	@November 17, 2021
👤 발표자	김선영

## 13.1 선언 및 기본 사용

여러 필드를 묶어서 하나의 구조체를 만듦. 다른 타입의 값들을 변수 하나로 묶어주는 기능

```
type 타입명 struct {  
    필드명 타입  
    ...  
    필드명 타입  
}
```

예시 학생 구조체

```
type Student struct {  
    Name string  
    Class int  
    No int  
    Score float64  
}
```

```
var a Student
```

- a는 Student의 필드들을 포함, a에 속한 각 필드에는 a.Name처럼 점 .을 찍어 접근 가능

```
package main  
  
import "fmt"  
  
type House struct { // House 구조체 정의
```

```

    Address string
    Size      int
    Price     float64
    Type      string
}

func main() {
    var house House          // House 구조체 변수 선언
    house.Address = "서울시 강동구 ..." // 각 필드값 초기화
    house.Size = 28
    house.Price = 9.8
    house.Type = "아파트"

    fmt.Println("주소:", house.Address)
    fmt.Printf("크기 %d평\n", house.Size)
    fmt.Printf("가격: %.2f억원\n", house.Price)
    fmt.Println("타입:", house.Type)
}

```

## 출력문

```

주소: 서울시 강동구 ...
크기 28평
가격: 98.00억원
타입: 아파트

```

## 13.2 구조체 변수 초기화

### 13.2.1 초깃값 생략

초깃값 생략 → 모든 필드가 기본값으로 초기화

```
var house House
```

- Address: ""
- Size: 0
- Price: 0.0
- Type: ""

### 13.2.2 모든 필드 초기화

```
var house House{"서울시 강동구", 28, 9.80, "아파트"}
```

- Address: "서울시 강동구"
- Size: 28
- Price: 9.80
- Type: "아파트"

```
var house House{  
    "서울시 강동구",  
    28,  
    9.80,  
    "아파트",  
}
```

### 13.2.3 일부 필드 초기화

```
var house House{Size: 28, Type: "아파트"}
```

- Address: ""
- Size: 28
- Price: 0.0
- Type: "아파트"

```
var house House{  
    Size: 28,  
    Type: "아파트",  
}
```

## 13.3 구조체를 포함하는 구조체

구조체의 필드로 다른 구조체 포함 가능. 일반적인 내장 타입처럼 포함하는 방법과 포함된 필드 방식.

### 13.3.1 내장 타입처럼 포함하는 방식

```

type User struct { // 일반 고객용 구조체
    Name string
    ID    string
    Age   int
}

type VIPUser struct { // VIP 고객용 구조체
    UserInfo User
    VIPLevel int
    Price    int
}

```

- VIP고객도 고객이므로 이름, ID, 연령 정보를 각각 선언하지 않고 User 구조체 활용.

```

package main

import "fmt"

type User struct { // 일반 고객용 구조체
    Name string
    ID    string
    Age   int
}

type VIPUser struct { // VIP 고객용 구조체
    UserInfo User
    VIPLevel int
    Price    int
}

func main() {
    user := User{"송하나", "hana", 23}
    vip := VIPUser{
        User{"화랑", "hwarang", 40},
        3,
        250,
    } // User를 포함한 VIPUser 구조체 변수 초기화

    fmt.Printf("유저: %s ID: %s 나이 %d\n", user.Name, user.ID, user.Age)
    fmt.Printf("VIP 유저: %s ID: %s 나이 %d VIP 레벨: %d VIP 가격: %d만원\n",
        vip.UserInfo.Name, // UserInfo 안의 Name
        vip.UserInfo.ID,   // UserInfo 안의 ID
        vip.UserInfo.Age,
        vip.VIPLevel, // VIPUser의 VIPLevel
        vip.Price,
    )
}

```

## 출력문

```
유저: 송하나 ID: hana 나이 23
VIP 유저: 화랑 ID: hwrang 나이 40 VIP 레벨: 3 VIP 가격: 25만원
```

### 13.3.2 포함된 필드 방식

vip에서 Name이나 ID와 같이 UserInfo 안에 속한 필드를 접근하려면 vip.UserInfo.Name과 같이 두 단계를 걸쳐 접근. 다른 구조체를 포함할 때 필드명 생략하면 .을 한 번만 찍어 접근 가능

```
package main

import "fmt"

type User struct { // 일반 고객용 구조체
    Name string
    ID   string
    Age  int
}

type VIPUser struct { // VIP 고객용 구조체
    User      // 필드명 생략
    VIPLevel int
    Price     int
}

func main() {
    user := User{"송하나", "hana", 23}
    vip := VIPUser{
        User{"화랑", "hwarang", 40},
        3,
        250,
    }

    fmt.Printf("유저: %s ID: %s 나이 %d\n", user.Name, user.ID, user.Age)
    fmt.Printf("VIP 유저: %s ID: %s 나이 %d VIP 레벨: %d VIP 가격: %d만원\n",
        vip.Name, // . 하나로 접근 가능
        vip.ID,
        vip.Age,
        vip.VIPLevel,
        vip.Price,
    )
}
```

## 출력문

유저: 송하나 ID: hana 나이 23  
VIP 유저: 화랑 ID: hwrang 나이 40 VIP 레벨: 3 VIP 가격: 25만원

- 구조체 안에 포함된 다른 구조체의 필드명을 생략하는 경우 → '포함된 필드'

## 필드 중복 해결

만약 포함된 필드 안에 속한 필드명과 상위 구조체의 필드명이 서로 겹치는 경우?

```
package main

import "fmt"

type User struct {
    Name string
    ID    string
    Age   int
    Level int // User의 Level 필드
}

type VIPUser struct {
    User // Level 필드를 갖는 구조체
    Price int
    Level int // VIPUser의 Level 필드
}

func main() {
    user := User{"송하나", "hana", 23, 10}
    vip := VIPUser{
        User{"화랑", "hwarang", 40, 10},
        250,
        3,
    }

    fmt.Printf("유저: %s ID: %s 나이 %d\n", user.Name, user.ID, user.Age)
    fmt.Printf("VIP 유저: %s ID: %s 나이 %d VIP 레벨: %d 유저 레벨:%d\n",
        vip.Name,
        vip.ID,
        vip.Age,
        vip.Level, // VIPUser의 Level
        vip.User.Level, // 포함된 구조체명을 쓰고 접근
    )
}
```

출력문

유저: 송하나 ID: hana 나이 23  
VIP 유저: 화랑 ID: hwarang 나이 40 VIP 레벨: 3 유저 레벨:10

- User 구조체에 포함된 Level 필드에 접근하려면 vip.User.Level로 다시 점 .을 찍어서 접근.

## 13.4 구조체 크기

구조체가 차지하는 메모리 크기?

```
type User struct {  
    Age int  
    Score float64  
}
```

```
var user User
```

User 구조체의 user 변수 선언 시 컴퓨터는 Age, Score 필드를 연속되게 담을 수 있는 메모리 공간을 찾아 할당. 총 16 크기가 필요.

### 13.4.1 구조체 값 복사

```
package main  
  
import "fmt"  
  
type Student struct {  
    Age    int // 대문자로 시작하는 필드는 외부로 공개  
    No     int  
    Score  float64  
}  
  
func PrintStudent(s Student) {  
    fmt.Printf("나이:%d 번호:%d 점수:%.2f\n", s.Age, s.No, s.Score)  
}  
  
func main() {  
    var student = Student{15, 23, 88.2}  
  
    // student 구조체 모든 필드가 student2 로 복사  
    student2 := student
```

```
PrintStudent(student2) // 함수 호출시에도 구조체 복사
}
```

## 출력문

```
나이:15 번호:23 점수:88.20
```

- 필드명이 대문자로 시작하는 경우 패키지 외부로 공개되는 필드

### 13.4.2 필드 배치 순서에 따른 구조체 크기 변화

```
package main

import (
    "fmt"
    "unsafe"
)

type User struct {
    Age    int32    // 4바이트
    Score float64 // 8바이트
}

func main() {
    user := User{23, 77.2}
    fmt.Println(unsafe.Sizeof(user))
}
```

## 출력문

```
16
```

- Age 타입 int → int32 (4바이트)
- 앞에서 배운대로라면 User 크기는 12바이트여야함. → 16바이트 출력.
- → 메모리 정렬

### 13.4.3 메모리 정렬

- 메모리 정렬이란 컴퓨터가 데이터에 효과적으로 접근하고자 메모리를 일정 크기 간격으로 정렬하는 것을 말함. 데이터가 레지스터 크기와 똑같은 크기로 정렬되어 있으면 더욱 효율



적으로 데이터를 읽어올 수 있다.

- 예) 64비트 컴퓨터. int64 데이터의 시작 주소가 100번지일 경우 100은 8의 배수가 아니므로 레지스터 크기 8에 맞게 정렬되어 있지 않음. 이럴 경우 데이터를 메모리에서 읽어올 때 성능을 손해보기 때문에 처음부터 프로그램 안에서 데이터를 만들 때 8의 배수인 메모리 주소에 데이터 할당. (104 번지)

```
type User struct {
    Age int32
    Score float64
}

var user User
```

- Age 4바이트, Score 8바이트. user의 시작 주소가 240번지이면 Age 시작 주소 240번지, Score의 시작 주소 244번지 → 8의 배수가 아니므로 성능 손해. 그래서 프로그램 언어에서 User 구조체를 할당할 때 Age와 Score 사이를 4바이트만큼 띄워서 할당
- → 메모리 패딩

#### 13.4.4 메모리 패딩을 고려한 필드 배치 방법

```
package main

import (
    "fmt"
    "unsafe"
)

type User struct {
    A int8 // 1바이트
    B int  // 8바이트
    C int8 // 1바이트
    D int  // 8바이트
    E int8 // 1바이트
}

func main() {
    user := User{1, 2, 3, 4, 5}
    fmt.Println(unsafe.Sizeof(user))
}
```

출력문

- User 구조체는 1바이트 필드 3개, 8바이트 필드 2개 → 19 바이트
- 하지만 실제 구조체 크기는 메모리 패딩 때문에 40바이트가 된다. 1 바이트 변수 모두에 7 바이트 씩 패딩

"8바이트보다 작은 필드는 8바이트 크기(단위)를 고려해서 몰아서 배치하자"

```
package main

import (
    "fmt"
    "unsafe"
)

type User struct {
    A int8 // 1바이트
    C int8 // 1바이트
    E int8 // 1바이트
    B int   // 8바이트
    D int   // 8바이트
}

func main() {
    user := User{1, 2, 3, 4, 5}
    fmt.Println(unsafe.Sizeof(user))
}
```

출력문

- 구조체 크기가 24바이트로 줄어듦. 40바이트보다 16바이트 절약
- 메모리 용량이 충분한 데스크톱이라면 패딩으로 인한 메모리 낭비를 크게 걱정하지 않아도 좋다. 메모리 공간이 작은 임베디드 하드웨어라면 패딩 고려하는 것이 좋음

### 13.5 프로그래밍에서 구조체의 역할

프로그래밍 역사는 객체 간 결합도(객체 간 의존관계)는 낮추고 연관있는 데이터 간 응집도를 올리는 방향으로 흘러왔다. 함수, 배열, 구조체 모두 응집도를 증가시키는 역할을 한다.

- 함수는 관련 코드 블록을 묶어서 응집도를 높이고 재사용성을 증가시킨다.
- 배열은 같은 타입의 데이터들을 묶어서 응집도를 높인다.
- 구조체는 관련된 데이터들을 묶어서 응집도를 높이고 재사용성을 증가시킨다.

구조체를 사용해서 관련 데이터들을 묶으면 프로그래머는 설계 과정에서 개별 데이터에 신경쓰지 않고 더 큰 범위에서 프로그램을 설계할 수 있다. 주요 구조체 위주로 설계하고 개별 데이터들은 나중에 구조체 안의 필드 형태로 추가/삭제할 수 있기 때문에 설계 과정에는 크게 신경쓰지 않아도 되는 원리이다.

프로그래머가 개별 데이터보다 큰 범위에서 생각함으로써 코딩의 중심이 개별 데이터의 조작/연산보다 구조체 간의 관계와 상호작용 중심으로 변화했다. 메서드, 인터페이스 개념이 추가되면서 객체지향 프로그래밍으로 발전했다.

## 연습문제

1.

```
type Product struct {
    Name string
    Price int
    ReviewScore float64
}
```

2. 200

8.7

3.

```
type Padding struct {
    A int8
    G int8
    D uint16
    F float32
    B int
    C float64
    E int
}
```