Hello, World

Задание 1: Базовая программа "Hello, World!"

1. Создайте папку `hello` в вашей рабочей директории

2. Создайте файл `hello.go` со следующим содержимым:

```go
package main

import (
    "fmt"
)

func hello_wrld() {
    fmt.Println("Hello, my name is Timur")
}
```

Рис.1: Задание 1

3. Запустите программу командой:

go run hello.go

Задание 2: Модифицированная программа с именем и датой

Вот модифицированная версия программы:

```go
package main

import (
    "fmt"
    "time"
)

func hello_time() {
    fmt.Println("Hello, my name is Timur")
    fmt.Println("Current date:", time.Now().Format("2006-01-02"))
}
```

Рис.2: Задание 2

Объяснение изменений:

1. Добавлен импорт пакета `time` для работы с датой

2. Использована переменная `name` для хранения вашего имени

3. `time.Now()` получает текущее время, а `Format()` задает формат вывода

4. Использован `fmt.Printf` для форматированного вывода

Особенности формата даты в Go:

- Go использует конкретные значения для форматирования даты:

  - `2006` - год

  - `01` - месяц

  - `02` - день

- Это может показаться странным, но это сделано для запоминания формата

Чтобы запустить модифицированную программу:

go run hello.go

Программа выведет :

Hello, my name is Timur

Сегодняшняя дата: 2025-06-21

Компиляция и запуск программы Go

Цель: Изучение процесса компиляции и запуска программ на Go, включая кросс-компиляцию для разных операционных систем.

1. Подготовка исходного кода

Создан файл `hello.go` со следующим содержимым:

Это простая программа, выводящая строку `"Hello, World!"` в консоль.

2. Задание 1: Компиляция и запуск программы

2.1. Запуск программы напрямую (без компиляции)

Команда:

go run hello.go

Вывод:

Hello, World!

Вывод: Программа успешно выполнилась, но исполняемый файл не создался.

2.2. Компиляция в исполняемый файл

Команда:

go build hello.go

Результат:

- В Linux/macOS создался файл `hello`.

- В Windows создался файл `hello.exe`.

2.3. Запуск исполняемого файла

-Linux/macOS:

./hello

- Windows:

hello.exe

Вывод:

Hello, World!

Вывод: Программа успешно скомпилировалась и запустилась.

3. Задание 2: Кросс-компиляция для разных ОС

3.1. Компиляция для Windows (64-bit)

Команда:

GOOS=windows GOARCH=amd64 go build -o hello.exe hello.go

Результат:

- Создался файл `hello.exe` (исполняемый файл Windows).

3.2. Компиляция для Linux (64-bit)

Команда:

GOOS=linux GOARCH=amd64 go build -o hello_linux hello.go

Результат:

- Создался файл `hello_linux` (исполняемый файл Linux).

3.3. Компиляция для macOS (64-bit, Intel/Apple Silicon)

Команда:

GOOS=darwin GOARCH=amd64 go build -o hello_macos hello.go

Результат:

- Создался файл `hello_macos` (исполняемый файл macOS).

3.4. Проверка скомпилированных файлов

После выполнения всех команд в папке появились 3 файла:

- `hello.exe` (Windows)
- `hello_linux` (Linux)
- `hello_macos` (macOS)

3.5. Тестирование на целевых системах

1. Windows:
   - Запуск:
     hello.exe
   - Вывод:
     Hello, World!
2. Linux:
   - Даем права на выполнение:
     chmod +x hello_linux
   - Запуск:
     ./hello_linux
   - Вывод:
     Hello, World!
3. macOS:
   - Даем права на выполнение:
     chmod +x hello_macos
   - Запуск:
     ./hello_macos
   - Вывод: Hello, World!

Вывод: Все исполняемые файлы работают корректно на своих платформах.

Переменные

Задание 1: Работа с переменными разных типов

1. Создана программа, объявляющая переменные основных типов:

- Целочисленные (`int`)
- Строковые (`string`)
- Логические (`bool`)
- Числа с плавающей точкой (`float64`)

```go
package main

import "fmt"

func main() {
    var age int = 25
    var name string = "Alice"
    var isStudent bool = true
    var height float64 = 1.75

    fmt.Println("Age:", age)
    fmt.Println("Name:", name)
    fmt.Println("Is student:", isStudent)
    fmt.Println("Height:", height)

    country := "Russia"
    temperature := 23.5
    sunny := false

    fmt.Println("Country:", country)
    fmt.Println("Temperature:", temperature)
    fmt.Println("Is sunny:", sunny)

    var defaultInt int
    var defaultString string
    var defaultBool bool
    var defaultFloat float64

    fmt.Println("Default int:", defaultInt)
    fmt.Println("Default string:", defaultString)
    fmt.Println("Default bool:", defaultBool)
    fmt.Println("Default float:", defaultFloat)
}
```

Рис.3: Задание 1

2. Продемонстрированы:

- Различные способы объявления переменных
- Краткий синтаксис инициализации с `:=`
- Значения по умолчанию для разных типов

Результат:

Программа успешно выводит значения всех объявленных переменных, демонстрируя особенности работы с разными типами данных в Go.

Задание 2: Использование математических констант

1. Объявлены константы для математических вычислений:

- π (pi)

- e (основание натурального логарифма)

- Золотое сечение

- √2 (квадратный корень из 2)

- ln(2) (натуральный логарифм 2)

```go
package main

import (
    "fmt"
    "math"
)

func main() {
    const pi = math.Pi
    const e = math.E
    const goldenRatio = 1.61803398875

    radius := 5.0
    circleArea := pi * math.Pow(radius, 2)
    fmt.Printf("Area of circle with radius %.2f: %.2f\n", radius, circleArea)

    x := 2.0
    exp := math.Pow(e, x)
    fmt.Printf("e^%.2f = %.4f\n", x, exp)

    a := 10.0
    b := a / goldenRatio
    fmt.Printf("For length %.2f, golden ratio gives %.2f\n", a, b)

    const sqrt2 = math.Sqrt2
    const ln2 = math.Ln2

    fmt.Println("Square root of 2:", sqrt2)
    fmt.Println("Natural logarithm of 2:", ln2)
}
```

Рис.4: Задание 2

2. Показано практическое применение констант:

- Вычисление площади круга

- Расчет экспоненты

- Пример с золотым сечением

Результат:

Программа демонстрирует корректное использование математических констант в различных вычислениях, показывая их практическую ценность.

Выводы

В ходе выполнения заданий:

1. Успешно освоены базовые принципы работы с переменными в Go

2. Изучены особенности объявления и инициализации переменных разных типов

3. Освоена работа с константами, включая математические константы из стандартной библиотеки

4. Получен практический опыт написания простых программ на Go

Базовые типы

Выполненные задания:

1. Демонстрация базовых типов данных

- Создана программа, показывающая все базовые типы Go с примерами значений:

  - Целые числа (`int`, `int8`-`int64`, `uint`, `uint8`-`uint64`, `rune`, `byte`)

  - Числа с плавающей точкой (`float32`, `float64`)

  - Комплексные числа (`complex64`, `complex128`)

  - Логический тип (`bool`)

  - Строки (`string`)

- Для каждого типа приведены характерные значения (максимальные/минимальные, примеры)

```
package main

import "fmt"

func main() {
    var i int = -42
    var i8 int8 = -128
    var i16 int16 = 32767
    var i32 int32 = -2147483648
```

```go
    var i64 int64 = 9223372036854775807
    var ui uint = 42
    var ui8 uint8 = 255
    var ui16 uint16 = 65535
    var ui32 uint32 = 4294967295
    var ui64 uint64 = 18446744073709551615
    var r rune = 'Я'
    var b byte = 255

    var f32 float32 = 3.14
    var f64 float64 = 3.141592653589793

    var c64 complex64 = 1 + 2i
    var c128 complex128 = 3 + 4i

    var isTrue bool = true
    var isFalse bool = false

    var s string = "Hello, Go!"
    var multiLine string = `Многострочная
строка`

    fmt.Println("Целые числа:")
    fmt.Println("int:", i)
    fmt.Println("int8:", i8)
    fmt.Println("int16:", i16)
    fmt.Println("int32:", i32)
    fmt.Println("int64:", i64)
    fmt.Println("uint:", ui)
    fmt.Println("uint8:", ui8)
    fmt.Println("uint16:", ui16)
    fmt.Println("uint32:", ui32)
    fmt.Println("uint64:", ui64)
    fmt.Println("rune:", r, string(r))
    fmt.Println("byte:", b)

    fmt.Println("\nЧисла с плавающей точкой:")
    fmt.Println("float32:", f32)
    fmt.Println("float64:", f64)

    fmt.Println("\nКомплексные числа:")
    fmt.Println("complex64:", c64)
    fmt.Println("complex128:", c128)

    fmt.Println("\nЛогические значения:")
    fmt.Println("bool true:", isTrue)
    fmt.Println("bool false:", isFalse)

    fmt.Println("\nСтроки:")
    fmt.Println("string:", s)
```

```
    fmt.Println("многострочная string:", multiLine)
}
```

Рис.5: Задание 1

2. Определение размеров типов данных

- Написана программа, измеряющая размер каждого типа с помощью `unsafe.Sizeof()`

- Показаны размеры всех базовых типов в байтах

- Отмечено, что размер `int`/`uint` зависит от архитектуры

- Указано, что для строк `unsafe.Sizeof()` возвращает размер структуры, а не длину содержимого

```
package main

import (
    "fmt"
    "unsafe"
)

func main() {
    var i int
    var i8 int8
    var i16 int16
    var i32 int32
    var i64 int64
    var ui uint
    var ui8 uint8
    var ui16 uint16
    var ui32 uint32
    var ui64 uint64
    var r rune
    var b byte

    var f32 float32
    var f64 float64

    var c64 complex64
    var c128 complex128

    var bl bool

    var s string

    fmt.Println("Размеры типов в байтах:")
    fmt.Println("int:", unsafe.Sizeof(i))
    fmt.Println("int8:", unsafe.Sizeof(i8))
```

```
    fmt.Println("int16:", unsafe.Sizeof(i16))
    fmt.Println("int32:", unsafe.Sizeof(i32))
    fmt.Println("int64:", unsafe.Sizeof(i64))
    fmt.Println("uint:", unsafe.Sizeof(ui))
    fmt.Println("uint8:", unsafe.Sizeof(ui8))
    fmt.Println("uint16:", unsafe.Sizeof(ui16))
    fmt.Println("uint32:", unsafe.Sizeof(ui32))
    fmt.Println("uint64:", unsafe.Sizeof(ui64))
    fmt.Println("rune:", unsafe.Sizeof(r))
    fmt.Println("byte:", unsafe.Sizeof(b))

    fmt.Println("\nЧисла с плавающей точкой:")
    fmt.Println("float32:", unsafe.Sizeof(f32))
    fmt.Println("float64:", unsafe.Sizeof(f64))

    fmt.Println("\nКомплексные числа:")
    fmt.Println("complex64:", unsafe.Sizeof(c64))
    fmt.Println("complex128:", unsafe.Sizeof(c128))

    fmt.Println("\nЛогический тип:")
    fmt.Println("bool:", unsafe.Sizeof(bl))

    fmt.Println("\nСтроки:")
    fmt.Println("string:", unsafe.Sizeof(s), "(размер структуры строки)")

    exampleString := "пример"
    fmt.Println("Длина строки 'пример' в байтах:", len(exampleString))
}
```

Рис.6: Задание 2

Вывод:

- Go предоставляет богатый набор числовых типов для оптимизации памяти

- Размер `int` соответствует разрядности системы (4 или 8 байт)

- Беззнаковые типы (`uint`) удваивают положительный диапазон значений

- `rune` и `byte` являются алиасами для `int32` и `uint8` соответственно

- Размер строки в Go - это размер дескриптора (16 байт на 64-битной системе), а не длины данных

Строки

Задание 1: Обработка строки

1. Подсчет длины строки – `len(str)`

2. Поиск подстроки – `strings.Contains(str, substr)`

3. Изменение регистра – `ToUpper()`, `ToLower()`

4. Доп. операции:

- `Count()` – подсчёт вхождений подстроки

- `ReplaceAll()` – замена подстроки

- `HasPrefix()`, `HasSuffix()` – проверка начала/конца строки

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    str := "Hello, World! This is Go Programming."

    length := len(str)
    fmt.Printf("1. Длина строки: %d символов\n", length)

    substr := "Go"
    contains := strings.Contains(str, substr)
    fmt.Printf("2. Строка содержит '%s': %t\n", substr, contains)

    upper := strings.ToUpper(str)
    lower := strings.ToLower(str)
    fmt.Println("3. В верхнем регистре:", upper)
    fmt.Println("   В нижнем регистре:", lower)

    fmt.Println("\nДополнительные операции:")
    fmt.Println("Количество слов 'is':", strings.Count(str, "is"))
    fmt.Println("Замена 'Go' на 'Golang':", strings.ReplaceAll(str, "Go", "Golang"))
    fmt.Println("Начинается с 'Hello':", strings.HasPrefix(str, "Hello"))
    fmt.Println("Заканчивается на '.':", strings.HasSuffix(str, "."))
}
```

Рис.7: Задание 1

Вывод: Программа успешно выполняет базовые операции со строками.

Задание 2: Разбиение предложения на слова

1. Разделение по пробелам – `strings.Fields()`

2. Разделение по запятым – `strings.Split(..., ",")`

3. Вывод слов – цикл `for range` с нумерацией.

```
package main

import (
    "fmt"
    "strings"
```

```
)

func main() {
   sentence := "Go is an open source programming language that makes it easy to build simple,
reliable, and efficient software."

   words := strings.Fields(sentence)

   fmt.Println("Слова в предложении:")
   for i, word := range words {
      fmt.Printf("%d: %s\n", i+1, word)
   }

   commaSentence := "apple,orange,banana,grape"
   fmt.Println("\nРазбиение строки по запятым:")
   fruits := strings.Split(commaSentence, ",")
   for _, fruit := range fruits {
      fmt.Println(fruit)
   }
}
```

Рис.8: Задание 2

Вывод: Программа корректно разбивает строку на слова и выводит их построчно.

Вывод:

- Обе программы используют пакет `strings`.

- Демонстрируются ключевые методы работы с **неизменяемыми** строками в Go.

- Код готов к использованию и масштабированию.

Срезы

Задание 1: Создание и динамическое добавление элементов

- Создан пустой срез `[]string`.

- Элементы добавлены динамически через `append()`.

- Все элементы выведены с индексами.

```
package main

import "fmt"

func main() {
   var mySlice []string

   mySlice = append(mySlice, "First")
   mySlice = append(mySlice, "Second")
```

```
    mySlice = append(mySlice, "Third")

    fmt.Println("Элементы среза:")
    for i, element := range mySlice {
        fmt.Printf("Индекс: %d, Значение: %s\n", i, element)
    }
}
```

Рис.9: Задание 2

Задание 2: Удаление элемента по индексу\

- Реализована функция `removeElement(slice []string, index int) []string`.

- Проверка на корректность индекса.

- Удаление через `append(slice[:index], slice[index+1:]...)`.

```
package main

import "fmt"

func removeElement(slice []string, index int) []string {
    if index < 0 || index >= len(slice) {
        fmt.Println("Неверный индекс")
        return slice
    }

    return append(slice[:index], slice[index+1:]...)
}

func main() {

    mySlice := []string{"First", "Second", "Third", "Fourth"}

    fmt.Println("Исходный срез:", mySlice)

    mySlice = removeElement(mySlice, 2)

    fmt.Println("Срез после удаления:", mySlice)
}
```

Рис.10: Задание 2

Вывод:

- Срезы в Go — гибкая альтернатива массивам с динамическим размером.

- `append()` расширяет срез, `copy()` копирует данные.

- Удаление элемента требует аккуратной работы с индексами и `append`.

- Оптимизация: задание ёмкости (`make([]T, length, capacity)`) снижает нагрузку от переаллокаций.

Map

Задание 1: Карта оценок студентов

- Реализация:

  - Создана карта `grades := make(map[string]int)` для хранения оценок (ключ: имя студента, значение: оценка).

  - Реализованы функции:

    - `addGrade`: добавляет оценку в карту (`grades[name] = grade`).

    - `findGrade`: ищет оценку по имени (возвращает `-1`, если студент не найден).

    - `removeGrade`: удаляет оценку (`delete(grades, name)`).

```go
package main

import "fmt"

func main() {
    grades := make(map[string]int)

    addGrade(grades, "Alice", 95)
    addGrade(grades, "Bob", 87)
    addGrade(grades, "Charlie", 91)

    fmt.Println("Alice's grade:", findGrade(grades, "Alice"))
    fmt.Println("Bob's grade:", findGrade(grades, "Bob"))

    fmt.Println("Dave's grade:", findGrade(grades, "Dave"))

    removeGrade(grades, "Bob")
    fmt.Println("After removal, Bob's grade:", findGrade(grades, "Bob"))

    fmt.Println("All grades:", grades)
}

func addGrade(grades map[string]int, name string, grade int) {
    grades[name] = grade
    fmt.Printf("Added grade for %s: %d\n", name, grade)
}

func findGrade(grades map[string]int, name string) int {
    grade, exists := grades[name]
    if !exists {
        return -1
    }
    return grade
}
```

```
func removeGrade(grades map[string]int, name string) {
   delete(grades, name)
   fmt.Printf("Removed grade for %s\n", name)
}
```

Рис.11: Задание 1

- Результат:

  - Успешное добавление, поиск и удаление элементов из карты.

Задание 2: Частота слов в тексте

- Реализация:

  - Функция `countWords`:

    - Разбивает текст на слова с учетом только букв и цифр (`strings.FieldsFunc`).

    - Приводит слова к нижнему регистру (`strings.ToLower`).

    - Подсчитывает частоту слов через карту `frequency := make(map[string]int)`.

```
package main

import (
   "fmt"
   "strings"
   "unicode"
)

func main() {
   text := `Go is an open source programming language that makes it easy to build simple,
   reliable, and efficient software. Go is a statically typed, compiled programming
   language designed at Google.`

   wordFrequency := countWords(text)

   fmt.Println("Word frequency:")
   for word, count := range wordFrequency {
      fmt.Printf("%s: %d\n", word, count)
   }
}

func countWords(text string) map[string]int {
   frequency := make(map[string]int)

   words := strings.FieldsFunc(text, func(r rune) bool {
      return !unicode.IsLetter(r) && !unicode.IsNumber(r)
   })
```

```
    for _, word := range words {
        lowerWord := strings.ToLower(word)
        frequency[lowerWord]++
    }

    return frequency
}
```

Рис.12: Задание 2

- Результат:

  - Для текста о языке Go выведена частота каждого слова:

Вывод:

- Карты в Go эффективны для хранения пар ключ-значение.

- Основные операции: добавление (`map[key] = value`), поиск (`value, exists := map[key]`), удаление (`delete(map, key)`).

- Применение: от простых справочников (оценки) до сложной аналитики (частотный анализ).

Циклы

Задание 1: Таблица умножения

- Использован вложенный цикл `for`

- Внешний цикл (1-10) - множители

- Внутренний цикл (1-10) - множимое

- Форматированный вывод с `fmt.Printf()`

```
package main

import "fmt"

func main() {
    for i := 1; i <= 10; i++ {
        fmt.Printf("Таблица умножения на %d:\n", i)
        for j := 1; j <= 10; j++ {
            fmt.Printf("%d x %d = %d\n", i, j, i*j)
        }
        fmt.Println()
    }
}
```

Рис.13: Задание 1

Результат:

- Выводит 10 таблиц умножения (от 1 до 10)

- Каждая таблица содержит 10 строк вида "a x b = c"

- Таблицы разделены пустыми строками

Задание 2: Простые числа до 100

- Цикл перебора чисел 2-100

- Для каждого числа проверка на простоту:

  - Проверка делителей до √n (оптимизация)

  - Флаг `isPrime` для отметки простых чисел

- Вывод только простых чисел

```go
package main

import "fmt"

func main() {
   fmt.Println("Простые числа до 100:")

   for num := 2; num <= 100; num++ {
      isPrime := true

      for i := 2; i*i <= num; i++ {
         if num%i == 0 {
            isPrime = false
            break
         }
      }

      if isPrime {
         fmt.Printf("%d ", num)
      }
   }

   fmt.Println()
}
```

Рис.14: Задание 2

Результат:

- Выводит все простые числа от 2 до 100 через пробел

- Корректно определяет простые числа (2, 3, 5, 7, 11... 97)

Вывод:

Оба решения демонстрируют эффективное использование циклов `for` в Go для решения типовых задач. Первая программа наглядно выводит таблицу умножения с помощью вложенных циклов, а вторая корректно находит простые числа, используя оптимизированную проверку делителей. Код соответствует идиомам Go, обеспечивая читаемость и производительность.

Условные операторы

Задание 1. Программа-калькулятор

- Функционал: выполняет базовые арифметические операции (+, -, *, /)

- Особенности:

  - Обработка ввода двух чисел и оператора

  - Проверка деления на ноль

  - Обработка неизвестных операторов

  - Вывод результата с точностью до 2 знаков после запятой

```
package main

import "fmt"

func main() {
   var a, b float64
   var operator string

   fmt.Print("Введите первое число: ")
   fmt.Scan(&a)

   fmt.Print("Введите оператор (+, -, *, /): ")
   fmt.Scan(&operator)

   fmt.Print("Введите второе число: ")
   fmt.Scan(&b)

   var result float64
   var err error = nil

   switch operator {
   case "+":
      result = a + b
   case "-":
      result = a - b
   case "*":
      result = a * b
   case "/":
      if b == 0 {
         err = fmt.Errorf("деление на ноль невозможно")
```

```
    } else {
        result = a / b
    }
  default:
    err = fmt.Errorf("неизвестный оператор")
  }

  if err != nil {
    fmt.Println("Ошибка:", err)
  } else {
    fmt.Printf("Результат: %.2f %s %.2f = %.2f\n", a, operator, b, result)
  }
}
```

Рис.15: Задание 1

Задание 2. Определитель дней недели

- Функционал: выводит название дня недели по его номеру (1-7)

- Особенности:

  - Простой ввод номера дня

  - Использование switch для ясности кода

  - Обработка неверных значений через default

```
package main

import "fmt"

func main() {
  var dayNumber int

  fmt.Print("Введите номер дня недели (1-7): ")
  fmt.Scan(&dayNumber)

  var dayName string

  switch dayNumber {
  case 1:
    dayName = "Понедельник"
  case 2:
    dayName = "Вторник"
  case 3:
    dayName = "Среда"
  case 4:
    dayName = "Четверг"
  case 5:
    dayName = "Пятница"
  case 6:
    dayName = "Суббота"
```

```
    case 7:
        dayName = "Воскресенье"
    default:
        dayName = "Неверный номер дня недели"
    }

    fmt.Println(dayName)
}
```

Рис.16: Задание 2

Вывод:

- В switch не требуется break после каждого case

- Переменные в блоках if/switch имеют ограниченную область видимости

- Конструкция switch предпочтительнее для множественных условий

Операторы

Задание 1: Арифметические операции

- Демонстрация всех основных арифметических операций:

  - Сложение (`+`)

  - Вычитание (`-`)

  - Умножение (`*`)

  - Деление (`/`) - целочисленное для целых чисел

  - Остаток от деления (`%`)

  - Инкремент (`++`) и декремент (`--`) в постфиксной форме

```
package main

import "fmt"

func main() {
    a := 10
    b := 3

    fmt.Println("Арифметические операции:")
    fmt.Printf("%d + %d = %d\n", a, b, a+b)
    fmt.Printf("%d - %d = %d\n", a, b, a-b)
    fmt.Printf("%d * %d = %d\n", a, b, a*b)
    fmt.Printf("%d / %d = %d (целочисленное деление)\n", a, b, a/b)
    fmt.Printf("%d %% %d = %d (остаток от деления)\n", a, b, a%b)

    a++
    fmt.Println("После a++:", a)
    a--
```

```
    fmt.Println("После a--:", a)
}
```
Рис.17: Задание 1

Задание 2: Проверка високосного года

- Реализована логика проверки с использованием:

  - Операторов сравнения (`==`, `!=`)

  - Логических операторов (`&&`, `||`)

- Протестированы граничные случаи (2000, 2004, 2100)

```
package main

import "fmt"

func isLeapYear(year int) bool {
    return (year%4 == 0 && year%100 != 0) || year%400 == 0
}

func main() {
    years := []int{2000, 2004, 2100, 2020, 2023}

    for _, year := range years {
        if isLeapYear(year) {
            fmt.Printf("%d - високосный год\n", year)
        } else {
            fmt.Printf("%d - не високосный год\n", year)
        }
    }
}
```
Рис.18: Задание 2

Вывод:

Оба задания успешно выполнены, демонстрируя работу:

- Арифметических операторов

- Операторов сравнения

- Логических операторов

- Особенностей Go (постфиксные инкремент/декремент)

Структуры

Задание 1: Структура "Студент"

- Создана структура `Student` с полями: `Name` (string), `Age` (int), `Course` (int), `GPA` (float64)

- Реализованы методы:
  - `NewStudent` - конструктор для создания нового студента
  - `PrintInfo` - вывод информации о студенте
  - `Promote` - повышение курса
  - `UpdateGPA` - обновление среднего балла

```go
package main

import "fmt"

type Student struct {
    Name   string
    Age    int
    Course int
    GPA    float64
}

func NewStudent(name string, age, course int, gpa float64) Student {
    return Student{
        Name:   name,
        Age:    age,
        Course: course,
        GPA:    gpa,
    }
}

func (s Student) PrintInfo() {
    fmt.Printf("Студент: %s\nВозраст: %d\nКурс: %d\nСредний балл: %.2f\n",
        s.Name, s.Age, s.Course, s.GPA)
}

func (s *Student) Promote() {
    s.Course++
    fmt.Printf("%s переведен на %d курс\n", s.Name, s.Course)
}

func (s *Student) UpdateGPA(newGPA float64) {
    s.GPA = newGPA
    fmt.Printf("Средний балл %s обновлен: %.2f\n", s.Name, s.GPA)
}

func main() {
    student := NewStudent("Иван Иванов", 20, 2, 4.5)

    student.PrintInfo()
    fmt.Println()

    student.Promote()
```

```
    student.UpdateGPA(4.7)
    fmt.Println()

    student.PrintInfo()
}
```

Рис.19: Задание 1

Задание 2: Структура "Автомобиль" с вложенной структурой "Двигатель"

- Создана вложенная структура: `Engine` внутри `Car`
- Поля двигателя: `Type` (string), `Power` (int), `Displacement` (float64)
- Реализованы методы:
  - `NewCar` - конструктор автомобиля
  - `PrintInfo` - вывод полной информации об автомобиле
  - `StartEngine` - имитация запуска двигателя

```
package main

import "fmt"

type Engine struct {
    Type        string
    Power       int     // в лошадиных силах
    Displacement float64 // объем в литрах
}

type Car struct {
    Make   string
    Model  string
    Year   int
    Engine Engine // вложенная структура
}

func NewCar(make, model string, year int, engineType string, power int, displacement float64)
Car {
    return Car{
        Make:  make,
        Model: model,
        Year:  year,
        Engine: Engine{
            Type:        engineType,
            Power:       power,
            Displacement: displacement,
        },
    }
}
```

```
func (c Car) PrintInfo() {
   fmt.Printf("Автомобиль: %s %s (%d год)\n", c.Make, c.Model, c.Year)
   fmt.Printf("Двигатель: %s, %.1f л, %d л.с.\n",
      c.Engine.Type, c.Engine.Displacement, c.Engine.Power)
}

func (c Car) StartEngine() {
   fmt.Printf("Заводим %s %s... Двигатель %s работает!\n",
      c.Make, c.Model, c.Engine.Type)
}

func main() {
   car := NewCar("Toyota", "Camry", 2022, "V6", 301, 3.5)

   car.PrintInfo()
   fmt.Println()

   car.StartEngine()
   fmt.Println()

   sportsCar := NewCar("Porsche", "911", 2023, "Flat-6", 379, 3.0)
   sportsCar.PrintInfo()
   sportsCar.StartEngine()
}
```

Рис.20: Задание 2

Вывод:

- Успешно реализованы структуры с различными типами полей

- Продемонстрированы принципы инкапсуляции и работы с методами структур

- Показана работа с вложенными структурами

- Примеры иллюстрируют основные возможности работы со структурами в Go

Функции

Задание 1: Функции для работы со срезами

1. Поиск элемента:

   - Функция `findElement` выполняет линейный поиск в срезе

   - Возвращает индекс элемента и флаг наличия

2. Сортировка:

   - Функция `sortSlice` создает копию среза и сортирует его

- Использует встроенную сортировку из пакета `sort`

- Сложность: O(n log n)

3. Фильтрация:

- Функция `filterSlice` создает новый срез с четными элементами

- Демонстрирует работу с динамическими срезами

- Сложность: O(n)

```
package main

import (
    "fmt"
    "sort"
)

func findElement(slice []int, target int) (int, bool) {
    for i, v := range slice {
        if v == target {
            return i, true
        }
    }
    return -1, false
}

func sortSlice(slice []int) []int {
    sorted := make([]int, len(slice))
    copy(sorted, slice)
    sort.Ints(sorted)
    return sorted
}

func filterSlice(slice []int) []int {
    var filtered []int
    for _, v := range slice {
        if v%2 == 0 {
            filtered = append(filtered, v)
        }
    }
    return filtered
}

func main() {
    numbers := []int{5, 3, 8, 1, 2, 7, 4, 6}

    if index, found := findElement(numbers, 8); found {
        fmt.Printf("Элемент 8 найден по индексу %d\n", index)
    } else {
        fmt.Println("Элемент не найден")
```

```
    }

    sorted := sortSlice(numbers)
    fmt.Println("Отсортированный срез:", sorted)

    filtered := filterSlice(numbers)
    fmt.Println("Отфильтрованный срез (четные числа):", filtered)
}
```

Рис.21: Задание 1

Задание 2: Поиск самой длинной строки

- Функция `findLongestString` принимает переменное число аргументов

- Обрабатывает случай пустого ввода

- Находит строку максимальной длины за один проход

```
package main

import "fmt"

func findLongestString(strings ...string) string {
    if len(strings) == 0 {
        return ""
    }

    longest := strings[0]
    for _, s := range strings[1:] {
        if len(s) > len(longest) {
            longest = s
        }
    }
    return longest
}

func main() {
    longest := findLongestString(
        "Привет",
        "Hello, World!",
        "Go",
        "Это тестовая строка",
        "Еще одна строка",
    )

    fmt.Println("Самая длинная строка:", longest)

    fmt.Println(findLongestString("один", "два", "три", "четыре"))
}
```

Рис.22: Задание 2

Вывод:

- Реализованы основные операции работы со срезами

- Продемонстрирована работа с вариативными функциями

- Показаны различные подходы к обработке коллекций

- Код соответствует идиоматическому стилю Go

Указатели

Задание 1: Функция обмена значений

Реализована функция `swap`, которая:

- Принимает два указателя на целые числа

- Обменивает значения переменных, используя разыменование указателей

- Демонстрирует работу с указателями для изменения оригинальных переменных

```go
package main

import "fmt"

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

func main() {
   x := 10
   y := 20

   fmt.Println("До обмена: x =", x, "y =", y)

   swap(&x, &y)

   fmt.Println("После обмена: x =", x, "y =", y)
}
```

Рис.23: Задание 1

Задание 2: Сравнение передачи параметров

Создана программа, показывающая разницу между:

1. Передачей по значению (`incrementByValue`):

   - Получает копию переменной

   - Изменения не влияют на оригинал

2. Передачей по указателю (`incrementByPointer`):

- Получает адрес переменной

- Может изменять оригинальное значение через разыменование

```go
package main

import "fmt"

func incrementByValue(a int) {
    a = a + 1
    fmt.Println("Внутри incrementByValue:", a)
}

func incrementByPointer(a *int) {
    *a = *a + 1
    fmt.Println("Внутри incrementByPointer:", *a)
}

func main() {
    val := 10

    fmt.Println("Исходное значение:", val)

    incrementByValue(val)
    fmt.Println("После incrementByValue:", val)

    incrementByPointer(&val)
    fmt.Println("После incrementByPointer:", val)

    fmt.Println("\nРазница между передачей по значению и по указателю:")
    fmt.Println("Передача по значению создает копию, оригинал не меняется")
    fmt.Println("Передача по указателю позволяет изменять оригинальную переменную")
}
```

Рис.24: Задание 2

Вывод:

- Указатели позволяют функциям изменять оригинальные переменные

- Передача по значению создает независимую копию данных

- Оператор `&` получает адрес переменной, `*` разыменовывает указатель

- Указатели особенно полезны для работы с большими структурами данных, чтобы избежать копирования

Методы

Задание 1: Структура "Студент"

Реализованы методы:

1. `CalculateAge()` - вычисляет возраст студента на основе года рождения

2. `GetStatus()` - определяет статус (отличник/хорошист/троечник) по среднему баллу

```go
package main

import (
    "fmt"
    "time"
)

type Student struct {
    Name      string
    BirthYear int
    Grades    []int
}

func (s Student) CalculateAge() int {
    currentYear := time.Now().Year()
    return currentYear - s.BirthYear
}

func (s Student) GetStatus() string {
    if len(s.Grades) == 0 {
        return "нет оценок"
    }

    sum := 0
    for _, grade := range s.Grades {
        sum += grade
    }
    average := float64(sum) / float64(len(s.Grades))

    switch {
    case average >= 4.5:
        return "отличник"
    case average >= 3.5:
        return "хорошист"
    default:
        return "троечник"
    }
}

func main() {
    student := Student{
        Name:      "Иван Иванов",
        BirthYear: 2000,
        Grades:    []int{5, 5, 4, 5, 4},
    }

    fmt.Printf("%s: возраст %d лет, статус - %s\n",
```

```
        student.Name,
        student.CalculateAge(),
        student.GetStatus())
}
```

Рис.25: Задание 1

Задание 2: Структура "Банковский счёт"

Реализованы методы:

1. `Deposit()` - пополнение счета с проверкой суммы

2. `Withdraw()` - снятие средств с проверкой баланса

3. `GetBalance()` - получение текущего баланса

```
package main

import (
    "errors"
    "fmt"
)

type BankAccount struct {
    owner   string
    balance float64
}

func (b *BankAccount) Deposit(amount float64) error {
    if amount <= 0 {
        return errors.New("сумма должна быть положительной")
    }
    b.balance += amount
    return nil
}

func (b *BankAccount) Withdraw(amount float64) error {
    if amount <= 0 {
        return errors.New("сумма должна быть положительной")
    }
    if b.balance < amount {
        return errors.New("недостаточно средств на счете")
    }
    b.balance -= amount
    return nil
}

func (b BankAccount) GetBalance() float64 {
    return b.balance
}

func main() {
```

```
    account := BankAccount{owner: "Петр Петров", balance: 1000.0}

    err := account.Deposit(500)
    if err != nil {
        fmt.Println("Ошибка пополнения:", err)
    }

    err = account.Withdraw(200)
    if err != nil {
        fmt.Println("Ошибка снятия:", err)
    }

    err = account.Withdraw(2000)
    if err != nil {
        fmt.Println("Ошибка снятия:", err)
    }

    fmt.Printf("Баланс счёта %s: %.2f\n", account.owner, account.GetBalance())
}
```

Рис.26: Задание 2

Вывод:

Оба задания успешно выполнены с демонстрацией:

- Разницы между получателями значений и указателей

- Принципов инкапсуляции данных в структурах

- Обработки ошибок в методах

- Практического применения методов для работы с данными структур

Интерфейсы

Задание 1: Интерфейс "Форма"

- Интерфейс `Shape` с методом `Area() float64`

- Две структуры: `Rectangle` и `Circle` с реализацией метода `Area()`

```
package main

import (
    "fmt"
    "math"
)

type Shape interface {
    Area() float64
}

type Rectangle struct {
```

```
    Width  float64
    Height float64
}

func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}

type Circle struct {
    Radius float64
}

func (c Circle) Area() float64 {
    return math.Pi * c.Radius * c.Radius
}

func main() {
    rect := Rectangle{Width: 5, Height: 3}
    circ := Circle{Radius: 2.5}

    shapes := []Shape{rect, circ}

    for _, shape := range shapes {
        fmt.Printf("Площадь фигуры: %.2f\n", shape.Area())
    }
}
```

Рис.27: Задание 1

Задание 2: Интерфейс "Транспорт"

- Интерфейс `Transport` с методами `Move()` и `Stop()`

- Три структуры: `Car`, `Bicycle` и `Train` с реализацией методов

```
package main

import "fmt"

type Transport interface {
    Move()
    Stop()
}

type Car struct {
    Model string
}

func (c Car) Move() {
    fmt.Printf("Автомобиль %s начал движение\n", c.Model)
}
```

```go
func (c Car) Stop() {
    fmt.Printf("Автомобиль %s остановился\n", c.Model)
}

type Bicycle struct {
    Brand string
}

func (b Bicycle) Move() {
    fmt.Printf("Велосипед %s начал движение\n", b.Brand)
}

func (b Bicycle) Stop() {
    fmt.Printf("Велосипед %s остановился\n", b.Brand)
}

type Train struct {
    Number int
}

func (t Train) Move() {
    fmt.Printf("Поезд №%d начал движение\n", t.Number)
}

func (t Train) Stop() {
    fmt.Printf("Поезд №%d остановился\n", t.Number)
}

func main() {
    vehicles := []Transport{
        Car{Model: "Toyota Camry"},
        Bicycle{Brand: "Stels"},
        Train{Number: 123},
    }

    for _, vehicle := range vehicles {
        vehicle.Move()
        vehicle.Stop()
        fmt.Println("---")
    }
}
```

Рис.28: Задание 2

Вывод:

- Интерфейсы в Go обеспечивают:

  - Четкое определение контрактов

  - Гибкость и полиморфизм

- Возможность расширения без модификации существующего кода