Алгоритмизация и программирование

4.5. Хранение данных в памяти

Глухих Михаил Игоревич
mailto: glukhikh@mail.ru

Системы счисления

- Система счисления – символический метод записи чисел; говоря иначе, способ представления чисел с помощью цифр (знаков, символов)
- Какие системы счисления бывают?

Десятичная система

- Примеры записи
  - $35164_{10}(=3*10^4+5*10^3+1*10^2+6*10+4)$

Десятичная система

- Примеры записи
  - $35164_{10}(=3*10^4+5*10^3+1*10^2+6*10+4)$
  - $6.48_{10}=6+4*10^{-1}+8*10^{-2}$

Десятичная система

- Примеры записи
  - $35164_{10}(=3*10^4+5*10^3+1*10^2+6*10+4)$
  - $6.48_{10}=6+4*10^{-1}+8*10^{-2}$
- Позиционная система счисления – вес каждой цифры зависит от ее положения (налево от десятичной точки – 1, 10, 10^2 и т.д.; направо – 10^{-1}, 10^{-2}, 10^{-3} и т.д.)

P-ичная система

- p – натуральное число не равное 1 – основание системы счисления

P-ичная система

- p – натуральное число не равное 1 – основание системы счисления
- Числа представляются в форме:
  - $X=x_n*p^n+x_{n-1}*p^{n-1}+\ldots+x_1*p+x_0+x_{-1}*p^{-1}+\ldots$, где x_i – цифры числа X в p-ичной системе; справедливо $0<=x_i<p$

P-ичная система

- p – натуральное число не равное 1 – основание системы счисления
- Числа представляются в форме:
  - $X=x_n*p^n+x_{n-1}*p^{n-1}+\ldots+x_1*p+x_0+x_{-1}*p^{-1}+\ldots$, где x_i – цифры числа X в p-ичной системе; справедливо $0<=x_i<p$

- Сокращенная форма записи
  - $X=x_nx_{n-1}x_1x_0.x_{-1}x_{-2}\ldots$

P-ичная система

- p – натуральное число не равное 1 – основание системы счисления
- Числа представляются в форме:
  - $X=x_n*p^n+x_{n-1}*p^{n-1}+\ldots+x_1*p+x_0+x_{-1}*p^{-1}+\ldots$, где x_i – цифры числа X в p-ичной системе; справедливо $0<=x_i<p$
- Сокращенная форма записи
  - $X=x_nx_{n-1}x_1x_0.x_{-1}x_{-2}\ldots$
- Например, $25_{10}=121_4$, так как $2*10+5=1*4^2+2*4+1$

Кто что использует?

- **Люди** в основном используют десятичную систему счисления. Основная причина – число пальцев на руках (удобно считать десятками)

Кто что использует?

- **Люди** в основном используют десятичную систему счисления. Основная причина – число пальцев на руках (удобно считать десятками)
- **Компьютеры**, как правило, хранят числа в **двоичной** системе счисления. Операции над числами также делаются в двоичной системе счисления. При выводе числа переводятся из двоичной системы в десятичную, при вводе – из десятичной в двоичную

Почему двоичная?

- Элементы с двумя четко выраженными состояниями проще в использовании и существенно дешевле, чем элементы с тремя и более состояниями

Почему двоичная?

- Элементы с двумя четко выраженными состояниями проще в использовании и существенно дешевле, чем элементы с тремя и более состояниями

- Основные операции в двоичной системе выполняются легче (ввиду простоты таблиц сложения и умножения)

Двоичная система в Котлине

```
val inBinary = 0b1001011 // 75
```

16-ная система в Котлине

```
val inHex = 0x4b // 75
```

16-ная система в Котлине

```
val inHex = 0x4b // 75
// 0xa = 10, 0xb = 11, 0xc = 12,
// 0xd = 13, 0xe = 14, 0xf = 15
```

16-ная система в Котлине

```
val inHex = 0x4b // 75
// 0xa = 10, 0xb = 11, 0xc = 12,
// 0xd = 13, 0xe = 14, 0xf = 15
// 0123456789abcdef - 16-ные цифры
```

16-ная система в Котлине

```
val inHex = 0x4b // 75
// 0xa = 10, 0xb = 11, 0xc = 12,
// 0xd = 13, 0xe = 14, 0xf = 15
// 0123456789abcdef - 16-ные цифры

// Из 16-ной в двоичную
//   4b
// 100 (4) 1011 (b) -> 100 1011
```

Числа в памяти компьютера

- Ячейки памяти состоят из отдельных двоичных разрядов.
- **Бит** – один двоичный разряд – единица измерения количества информации
- Восемь (2^3) **бит** образуют **байт**
- Сколько байт в килобайте?

Числа в памяти компьютера

- Ячейки памяти состоят из отдельных двоичных разрядов.
- **Бит** – один двоичный разряд – единица измерения количества информации
- Восемь (2^3) **бит** образуют **байт**
- Сколько байт в килобайте?
- $2^{10}=1024$ байт образуют **килобайт** (Кбайт)
  - 2^{10} Кбайт=1 **Мегабайт**
  - 2^{10} Мбайт=1 **Гигабайт**
  - 2^{10} Гбайт=1 **Терабайт**

Целые числа

- Int → 32 бита = 4 байта

Целые числа

- Int → 32 бита = 4 байта
  - 32 = 1 (знак) + 31 (модуль)

Целые числа

- Int → 32 бита = 4 байта
  - 32 = 1 (знак) + 31 (модуль)
  - x >= 0 : знак = 0, модуль = |x|

Целые числа

- Int → 32 бита = 4 байта
  - 32 = 1 (знак) + 31 (модуль)

  - x >= 0 : знак = 0, модуль = x
  - x < 0 : знак = 1, модуль = 2^{31} - |x|

Целые числа

- Int → 32 бита = 4 байта
  - 32 = 1 (знак) + 31 (модуль)

  - x >= 0 : знак = 0, модуль = x
  - x < 0 : знак = 1, модуль = 2^{31} - |x|

- Byte → 8 бит = 1 байт
- Short → 16 бит = 2 байта

Целые числа

- Int → 32 бита = 4 байта
  - 32 = 1 (знак) + 31 (модуль)

  - x >= 0 : знак = 0, модуль = x
  - x < 0 : знак = 1, модуль = 2^{31} - |x|

- Byte → 8 бит = 1 байт
- Short → 16 бит = 2 байта
- Long → 64 бита = 8 байт
  - `val longVal = 12345678901234L`

Вещественные числа

- Double → 64 бита = 8 байт

Вещественные числа

- Double → 64 бита = 8 байт
  - 64 = 11 (порядок) + 53 (мантисса)
  - Мантисса * $2^{\text{порядок}}$
  - |Мантисса| in 0.5 .. 1

Вещественные числа

- Double → 64 бита = 8 байт
  - 64 = 11 (порядок) + 53 (мантисса)
  - Мантисса * $2^{\text{порядок}}$
  - |Мантисса| in 0.5 .. 1
- Float → 32 бита = 4 байта

Вещественные числа

- Double → 64 бита = 8 байт
  - 64 = 11 (порядок) + 53 (мантисса)
  - Мантисса * $2^{\text{порядок}}$
  - |Мантисса| in 0.5 .. 1
- Float → 32 бита = 4 байта
  - 32 = 8 (порядок) + 24 (мантисса)

Вещественные числа

- Double → 64 бита = 8 байт
  - 64 = 11 (порядок) + 53 (мантисса)
  - Мантисса * $2^{\text{порядок}}$
  - |Мантисса| in 0.5 .. 1
- Float → 32 бита = 4 байта
  - 32 = 8 (порядок) + 24 (мантисса)
  - `val floatVal = 3.14159F`

Символы

- Char → 16 бит = 2 байта

Символы

- Char → 16 бит = 2 байта
  - Unicode: коды от 0 до $2^{16}-1$

Символы

- Char → 16 бит = 2 байта
  - Unicode: коды от 0 до $2^{16}-1$
- Int or Char x 2 → 32 бит = 4 байта
  - Unicode: коды от 0 до $2^{32}-1$

Символы

- Char → 16 бит = 2 байта
  - Unicode: коды от 0 до $2^{16}-1$
- Int or Char x 2 → 32 бит = 4 байта
  - Unicode: коды от 0 до $2^{32}-1$
- Спецсимволы
  - `'\n' '\r' '\t' '\'' '\"' '\\' '\$' '\uFF00'`

Значения и ссылки (JVM)

- Хранение значений
  - Переменная -> Ячейка памяти -> Значение

Значения и ссылки (JVM)

- Хранение значений
  - Переменная -> Ячейка памяти -> Значение
  - Используется (не всегда!):
    Byte, Short, Int, Long, Float, Double, Char, Boolean

Значения и ссылки (JVM)

- Хранение значений
  - Переменная -> Ячейка памяти -> Значение
  - Используется (не всегда!):
    Byte, Short, Int, Long, Float, Double, Char, Boolean
- Хранение ссылок
  - Переменная -> Ячейка памяти -> Ссылка на кучу!

Значения и ссылки (JVM)

- Хранение значений
  - Переменная -> Ячейка памяти -> Значение
  - Используется (не всегда!):
    Byte, Short, Int, Long, Float, Double, Char, Boolean
- Хранение ссылок
  - Переменная -> Ячейка памяти -> Ссылка на кучу!
  - Используется: всеми остальными типами

Значения и ссылки (JVM)

- Хранение значений
  - Переменная -> Ячейка памяти -> Значение
  - Используется (не всегда!):
    Byte, Short, Int, Long, Float, Double, Char, Boolean
- Хранение ссылок
  - Переменная -> Ячейка памяти -> Ссылка на кучу!
  - Куча = Heap = динамическая память = большой участок памяти, занимаемый и освобождаемый программой по мере необходимости

Куча

```
val list = listOf(1, 2, 3)
```

Куча

```
val list = listOf(1, 2, 3)
// Выделяется место в heap с номером 1
```

Куча

```
val list = listOf(1, 2, 3)
// Выделяется место в heap с номером 1
// Там сохраняется список
```

Куча

```
val list = listOf(1, 2, 3)
// Выделяется место в heap с номером 1
// Там сохраняется список
// А в ячейке list сохраняется номер 1
```

Куча и ссылки

```
fun foo() {
    // [1, 2, 3] хранится в участке кучи с номером 1,
    // а хранит номер 1
    val a = listOf(1, 2, 3)
}
```

Куча и ссылки

```
fun foo() {
    // [1, 2, 3] хранится в участке кучи с номером 1,
    // a хранит номер 1
    val a = listOf(1, 2, 3)
    // [4, 5] хранится в участке кучи с номером 2,
    // b хранит номер 2
    var b = listOf(4, 5)
}
```

Куча и ссылки

```
fun foo() {
    // [1, 2, 3] хранится в участке кучи с номером 1,
    // a хранит номер 1
    val a = listOf(1, 2, 3)
    // [4, 5] хранится в участке кучи с номером 2,
    // b хранит номер 2
    var b = listOf(4, 5)
    // Присваивание ссылок: b теперь хранит номер 1
    b = a
}
```

Куча и ссылки

```
fun foo() {
    // [1, 2, 3] хранится в участке кучи с номером 1,
    // a хранит номер 1
    val a = listOf(1, 2, 3)
    // [4, 5] хранится в участке кучи с номером 2,
    // b хранит номер 2
    var b = listOf(4, 5)
    // Присваивание ссылок: b теперь хранит номер 1
    b = a
    // Участок 2 не используется
}
```

Куча и ссылки

```
fun foo() {
    // [1, 2, 3] хранится в участке кучи с номером 1,
    // a хранит номер 1
    val a = listOf(1, 2, 3)
    // [4, 5] хранится в участке кучи с номером 2,
    // b хранит номер 2
    var b = listOf(4, 5)
    // Присваивание ссылок: b теперь хранит номер 1
    b = a
    // Участок 2 не используется
    // Будет освобождён сборщиком мусора
    // (Garbage Collector)
}
```

Неизменяемые и изменяемые ссылочные типы

- List<T>, String = неизменяемые

Неизменяемые и изменяемые ссылочные типы

- List<T>, String = неизменяемые

```
fun foo() {
    // Alpha: участок с номером 1
    val a = "Alpha"
    // Beta: участок с номером 2
    var b = "Beta"
    // Тоже номер 2
    val c = b
    // Формируем Alpha + Beta = AlphaBeta:
    // участок с номером 3
    b = a + b
}
```

Неизменяемые и изменяемые ссылочные типы

- MutableList<T>, Array<T> = изменяемые

Неизменяемые и изменяемые ссылочные типы

- MutableList<T>, Array<T> = изменяемые

```
fun foo() {
    // Участок с номером 1
    val a = mutableListOf(1, 2, 3)
    // Тоже номер 1
    val b = a
    // Изменение содержимого участка
    // с номером 1: теперь это [1, 2, 5]
    b[2] = 5
    println(a[2]) // ???
}
```

Неизменяемые и изменяемые ссылочные типы

- MutableList<T>, Array<T> = изменяемые

```
fun foo() {
    // Участок с номером 1
    val a = mutableListOf(1, 2, 3)
    // Тоже номер 1
    val b = a
    // Изменение содержимого участка
    // с номером 1: теперь это [1, 2, 5]
    b[2] = 5
    println(a[2]) // 5 (!)
}
```

Сравнение ссылок

- a === b, a !== b :
  сравнение **ссылок** (номеров)

Сравнение ссылок

- a === b, a !== b :
  сравнение **ссылок** (номеров)

- a == b, a != b :
  сравнение **содержимого**

Сравнение ссылок

- a === b, a !== b :
  сравнение **ссылок** (номеров)

- a == b, a != b :
  сравнение **содержимого**
  - Не определено для Array<T>

Сравнение ссылок

- a === b, a !== b :
  сравнение **ссылок** (номеров)

- a == b, a != b :
  сравнение **содержимого**
  - Не определено для Array<T>

```
fun foo() {
    val a = arrayOf(1, 2)
    val b = arrayOf(1, 2)
    println(a == b)  // ???
    println(a === b) // ???
}
```

Сравнение ссылок

- a === b, a !== b :
  сравнение **ссылок** (номеров)

- a == b, a != b :
  сравнение **содержимого**
  - Не определено для Array<T>

```
fun foo() {
    val a = arrayOf(1, 2)
    val b = arrayOf(1, 2)
    println(a == b)  // false
    println(a === b) // false
}
```

Стек JVM

- Хранит переменные, параметры, ...
- Расширяется и сужается по мере работы функций

Стек JVM

- Хранит переменные, параметры, ...
- Расширяется и сужается по мере работы функций

```
fun main(args: Array<String>) {
    // Stack: main::args
    foo(8)
}
```

Стек JVM

- Хранит переменные, параметры, ...
- Расширяется и сужается по мере работы функций

```
fun foo(n: Int): Int {
    // Stack: main::args, foo::n
    return bar(n / 2, n * 2)
}
fun main(args: Array<String>) {
    // Stack: main::args
    foo(8)
}
```

Стек JVM

- Хранит переменные, параметры, ...
- Расширяется и сужается по мере работы функций

```
fun bar(x: Int, y: Int): Int {
    // Stack: main::args, foo::n, bar::x, bar::y
    val z = x + y
    // Stack: main::args, foo::n, bar::x, bar::y, bar::z
    return z
}
fun foo(n: Int): Int {
    // Stack: main::args, foo::n
    return bar(n / 2, n * 2)
}
fun main(args: Array<String>) {
    // Stack: main::args
    foo(8)
}
```

Память JVM в целом

- Куча
- Стек
- Функции / Классы
- Константы