Курс «Алгоритмы и алгоритмические языки»
1 семестр 2013/2014

Лекция 9

Строки

```
#include <stdio.h>
#include <string.h>

int main (void)
{
  char string1[80], string2[80], smp[3];

  fgets (string1, 80, stdin);
  string1[strlen (string1)-1] = '\0';


  <...>
}
```

Возврат из функции

- Возврат из функции в точку вызвавшей ее функции, следующей за точкой вызова функции, осуществляется:
  - либо при выполнении оператора **`return`**,
  - либо после выполнения последнего оператора функции, если она не содержит оператора **`return`**.

```
#include <string.h>
#include <stdio.h>
void print_str_reverse (char *s)
{
  register int i;
  for (i = strlen (s) – 1; i >= 0; i--)
    putchar (s[i]);
}
```

- Если тип функции не **`void`**, то в ее теле должен быть хотя бы один оператор **`return`** с возвращаемым значением
- Если у функции несколько операторов **`return`**, возврат осуществляется **немедленно** по тому из них, который будет выполнен первым.

Результат выполнения функции

- Все функции, кроме тех, которые относятся к типу **`void`**, возвращают значение, которое определяется выражением в операторе **`return`**.
- Помимо вычисления возвращаемого значения, функция может изменять значения переменных вызывающей функции (по указателю), а также изменять значения глобальных переменных.
- Результаты вызова функции, не связанные непосредственно с вычислением возвращаемых значений, составляют *побочный эффект* функции.
- Выделяют следующие виды функций:
  (1) Функции, которые выполняют операции над своими аргументами с единственной целью – вычислить возвращаемое значение.
  (2) Функции, которые обрабатывают данные и возвращают значение, которое показывает, успешно ли была выполнена эта обработка.
  (3) Функции, возвращающие несколько значений (через указатели-аргументы и через возвращаемое значение).
  (4) Функции, не возвращающие значений. Все такие функции имеют тип **`void`**.

Результат выполнения функции

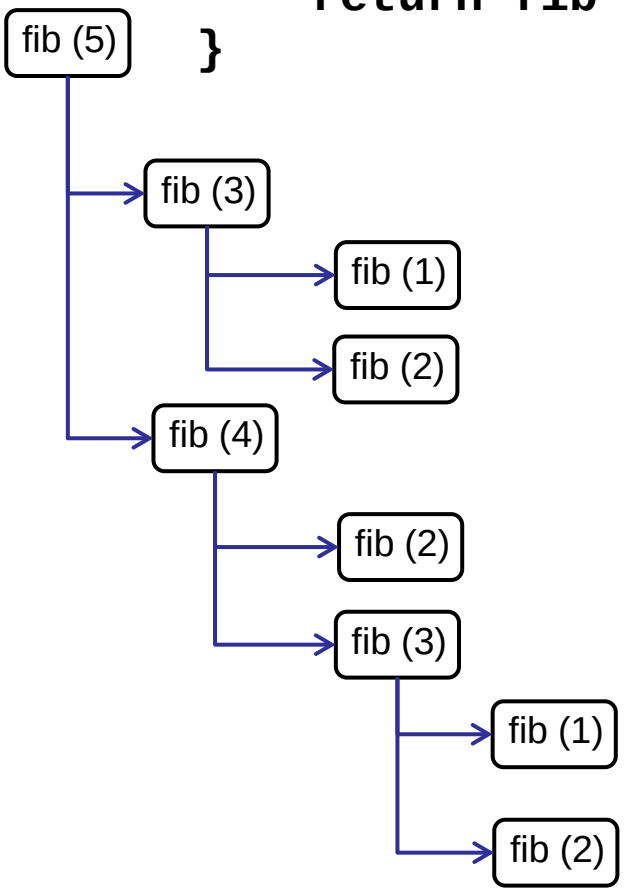
- Возвращаемым значением может быть указатель. Требуется, чтобы в объявлении такой функции тип возвращаемого указателя был объявлен точно: нельзя объявлять возвращаемый тип как **`int *`**, если функция возвращает указатель типа **`char *`**.
- Пример функции, возвращающей указатель (поиск первого вхождения символа **`c`** в строку **`s`**):

```
char *match (char c, char *s)
{
  while (c != *s && *s)
    s++;
  return s;
}
```

Рекурсия

- В языке Си функция может быть *рекурсивной*, т.е. вызывать саму себя:

```
int fib (int n)
{
  if (n == 1 || n == 2)
    return 1;
  else
    return fib (n – 2) + fib (n – 1);
}
```



Рекурсия

- Рекурсивные функции часто неэффективны по сравнению с их нерекурсивными вариантами:

```
int fibn (int n) {
  int i, g, h, fb;
  if (n == 1 || n == 2)
    return 1;
  else
    for (i = 2, g = h = 1; i < n; i++) {
      fb = g + h;
      h = g;
      g = fb;
    }
  return fb;
}
```

- Функция **`fib`** работает за экспоненциальное время и линейную память,
функция **`fibn`** – за линейное время и константную память.

Хвостовая рекурсия

- *Хвостовая рекурсия* (tail recursion) – рекурсивный вызов в самом конце функции. Как правило, этот вызов может быть оптимизирован компилятором в цикл.

```
int fact (int n) {
  if (n == 0)
    return 1;
  else
    return n*fact (n-1);
}
```

```
int fact (int n) {
  return tfact (n, 1);
}
int tfact (int n, int acc) {
  if (n == 0)
    return acc;
  return tfact (n-1, n*acc);
}
```

```
int fact (int n) {
  int t_n = n, t_acc = 1;

/* tfact встроена в fact и оптимизирована в цикл */
start:
  if (t_n == 0)
    return t_acc;
  t_acc = t_n * t_acc;
  t_n = t_n – 1;
  goto start;
}
```

Ключевое слово `inline`: подставляемые функции (C99)

```
#include <stdio.h>
inline static int max (int a, int b)
{
  return a > b ? a : b;
}
int main (void)
{
  int x = 5, y = 17;
  printf ("Наибольшим из чисел %d и %d является %d\n",
           x, y, max (x, y));
  return 0;
}
```

◈ При обычной реализации **`inline`** приведенная программа эквивалентна:

```
#include <stdio.h>
inline static int max (int a, int b)
{
  return a > b ? a : b;
}
int main (void)
{
  int x = 5, y = 17;
  printf ("Наибольшим из чисел %d и %d является %d\n",
           x, y, (x > y ? x : y));
  return 0;
}
```

Указатели на функцию

- Каждая функция располагается в памяти по определенному адресу. Адресом функции является ее точка входа (при вызове функции управление передается именно на эту точку).
- Присвоив значение адреса функции переменной типа указатель, получим указатель на функцию.
- Указатель функции можно использовать вместо ее имени при вызове этой функции. Указатель «лучше» имени тем, что его можно передавать другим функциям в качестве их аргумента.
- Имя функции **`f()`** без скобок и аргументов (**`f`**) по определению является указателем на функцию **`f()`** (аналогия с массивом).

```
int (*pf) (const char*, const char*);
char *s1, *s2;
int x = (*pf) (s1, s2);
int y = pf (s2, "string constant");
```

Указатели на функцию

- ***Пример***. Сравнение двух строк символов, введенных пользователем (функция **check()**).

```
#include <stdio.h>
#include <string.h>

static void check (char *a, char *b,
                   int (*pf) (const char*, const char*)) {
  printf ("Проверка на совпадение: ");
  if (! pf (a, b))
    printf ("равны\n");
  else
    printf ("не равны\n");
}

int main (void) {
  char s1[80], s2[80];

  printf ("Введите две строки \n");
  fgets (s1, sizeof (s1), stdin); s1[strlen (s1) - 1] = 0;
  fgets (s2, sizeof (s2), stdin); s2[strlen (s2) - 1] = 0;
  check (s1, s2, strcmp);
  return 0;
}
```

- Объявление **`int (*p)(const char *, const char *);`** сообщает компилятору, что **`p`** – указатель на функцию, имеющую два параметра типа **`const char *`** и возвращающую значение типа **`int`**.
- Скобки вокруг **`*p`** нужны, так как операция **`*`** имеет более низкий приоритет, чем **`()`**: если написать **`int *p(...)`**, получится, что объявлен не указатель на функцию, а функция **`p`**, которая возвращает указатель на целое.
- **`(*cmp)(a, b)`** эквивалентно **`cmp(a, b)`.**
- У функции **`check`** три параметра: два указателя на тип **`char`** и указатель на функцию **`pf`**. Указатель **`pf`** и функция **`strcmp`** имеют одинаковый формат, что позволяет использовать имя функции в качестве аргумента, соответствующего параметру **`pf`**.
- В данном случае использование указателя на функцию позволяет не менять программу сравнения, и тем самым получается более общий алгоритм.

```
int compvalues (const char *a, const char *b) {
  return atoi (a) != atoi (b);
}
```

- Массивы указателей на функцию: гибкая обработка событий

Вычисления с плавающей точкой

- Предпосылки: дробные двоичные числа
- Стандарт арифметики с плавающей точкой IEEE 754: Определение
- Пример и свойства
- Округление, сложение, умножение
- Плавающие типы языка Си
- Выводы

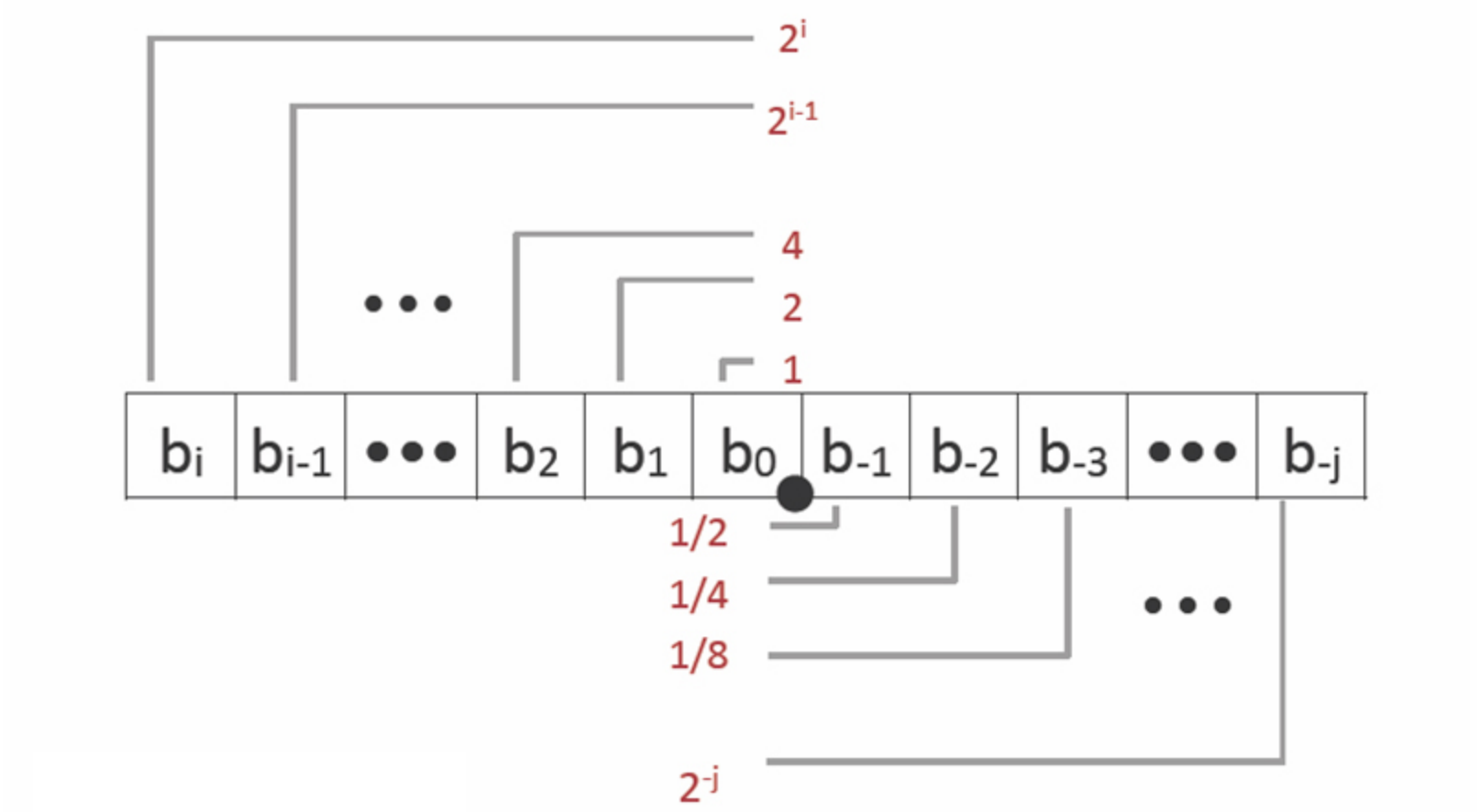
Дробные двоичные числа

- Что такое 1011.101_2 ?

$$1\times 2^3+0\times 2^2+1\times 2^1+1\times 2^0+1\times 2^{-1}+0\times 2^{-2}+1\times 2^{-3}=$$

$$=11\frac{5}{8}=11.625$$

Дробные двоичные числа



- Черное пятнышко – двоичная точка
- Биты слева от точки умножаются на положительные степени 2
- Биты справа от точки умножаются на отрицательные степени 2

Дробные двоичные числа

- $0.111111\ldots_2 = 1.0 - \varepsilon\ (\varepsilon \to 0)$, так как

$$1/2 + 1/4 + 1/8 + \ldots + 1/2^n + \ldots \to 1 \quad \text{при } n \to \infty$$

- Точно можно представить только числа вида $x/2^k$

- Остальные рациональные числа представляются периодическими двоичными дробями:

$$1/5 = 0.(0011)_2$$

- Иррациональные числа представляются апериодическими двоичными дробями и могут быть представлены только приближенно

Представление чисел с плавающей точкой (IEEE 754)

- Числа с плавающей точкой представляются в нормализованной форме: $(-1^s)\ M\ 2^e$
  - s – код знака числа (он же знак мантиссы)
  - M – мантисса ($1 \le M < 2$)
  - e – (двоичный) порядок

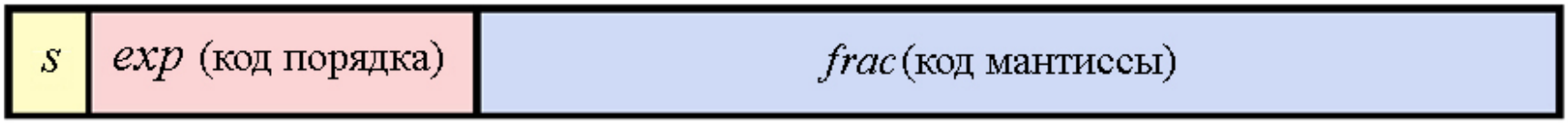
- Первая цифра мантиссы в нормализованном представлении всегда 1. В стандарте принято решение не записывать в представление числа эту единицу (тем самым мантисса как бы увеличивается на разряд).

  Экономия связана с тем, что в представление числа записывается не M, а $frac = M - 1$

Представление чисел с плавающей точкой

- Чтобы не записывать отрицательных чисел в поле порядка, вводится ***смещение*** $bias = 2^{k-1} - 1$, где k – количество бит в поле для записи порядка, и вместо порядка e записывается код порядка exp, связанный с e соотношением $e = exp - bias$.

- Нормализованное число $(-1^s)\ M\ 2^e$ упаковывается в машинное слово (структуру) с полями s, $frac$ и exp



Ширина поля s всегда равна 1.

Ширина полей exp и $frac$ зависит от точности числа

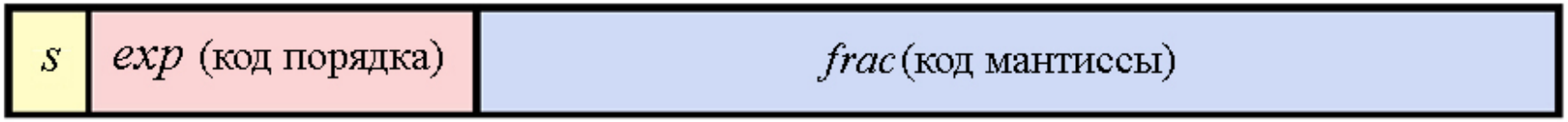
Представление чисел с плавающей точкой

- Одинарная точность (32 бита):

8 бит 23 бита

$bias = 127;$ $-126 \le e \le 127;$ $1 \le exp \le 254$

- Двойная точность (64 бита):



11 бит 52 бита

$bias = 1023;$ $-1022 \le e \le 1023;$ $1 \le exp \le 2046$

- Повышенная точность (80 бит):

15 бит 64 бита

Представление чисел с плавающей точкой

- Пример
  - **Значение** `float f = 15213.0`

    15213_{10} = 11101101101101_2 =

    $1.1101101101101_2 \times 2^{13}$
  - **Значащая часть**

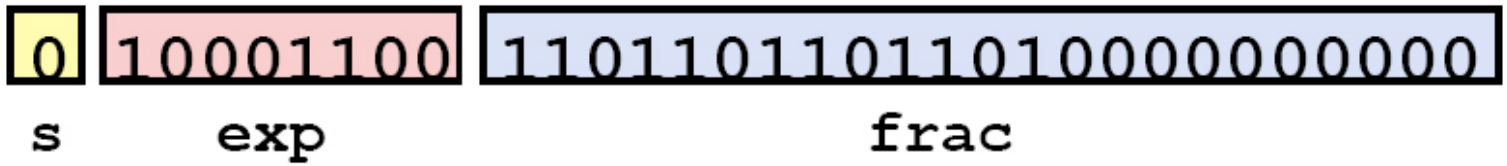
    $M = 1.\underline{1101101101101}_2$,

    $frac = \underline{1101101101101}0000000000_2$
  - **Порядок**

    $e = 13$

    $bias = 127$

    $exp = 140 = 10001100_2$
  - **Результат**



Представление нуля

- Для типа **`float`** код порядка **`exp`** изменяется от **`00000001`** до **`11111110`**
  (значению **`00000001`** соответствует порядок **`e = - 126`**,
  значению **`11111110`** – порядок **`e = 127`**)

- Код **`exp = 00000000, frac = 000…0`**
  представляет нулевое значение; в зависимости от значения знакового разряда **`s`** это либо **`+0`** либо **`-0`**

- А какое значение представляет код
  `exp = 00000000, frac ≠ 000…0`?

Большие числа

Пусть **exp = 111…1**

- если при этом **frac = 000…0**, то коду будет соответствовать значение ∞ (со знаком **s**)
- если же **frac ≠ 000…0**, то код не будет представлять никакое число («значение», представляемое таким кодом, так и называется: «не число» – **NaN** – Not a number)

Денормализованные числа

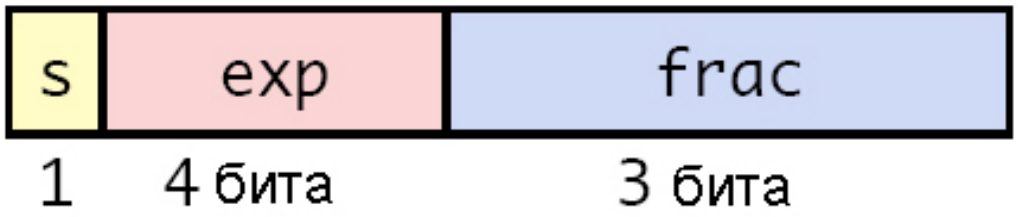
- Это числа, представляемые кодами

  exp = 00000000, frac ≠ 000…0

- **exp** вносит в значение такого числа постоянный вклад 2^{-k-2},

  frac меняется от **000…01** до **111…1 и** рассматривается уже не как мантисса, а как значение, умножаемое на **exp**

- Рассмотрим это на модельном примере:

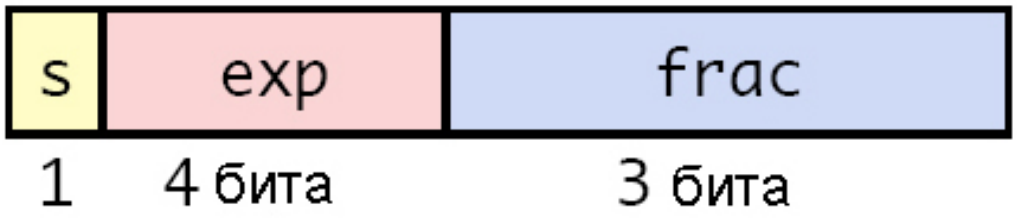


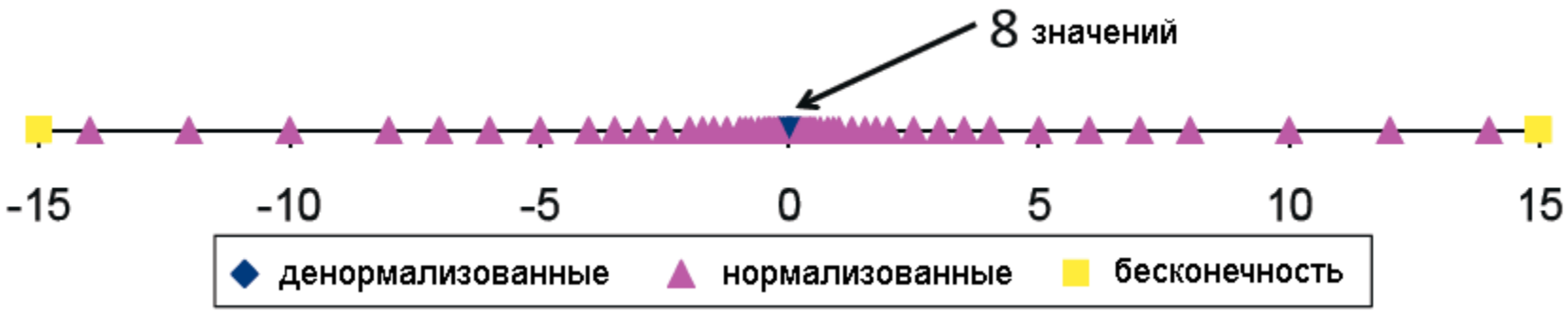
8-разрядные числа с плавающей точкой (положительные)

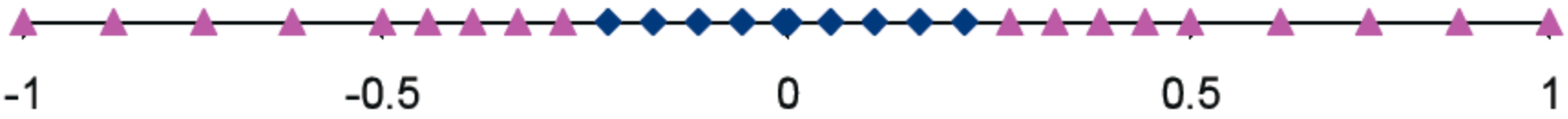
1 4 бита 3 бита

	s	exp	frac	E	Value	
Ненормализованные числа	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	Близкие к 0
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	Наибольшее ненормализованное
Нормализованные числа	0	0001	000	-6	8/8*1/64 = 8/512	Наименьшее нормализованное
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	Ближайшее к 1 снизу
	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	Ближайшее к 1 сверху
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	Наибольшее нормализованное
	0	1111	000	n/a	inf	

8-разрядные числа с плавающей точкой







Центральная область более крупно

Важные частные случаи

	exp	frac	Численное значение
◈ Нуль	00…00	00…00	0.0
◈ Наим. положит. денорм.	00…00	00…01	
♦ float ≈ 1.4×10^{-45}			$2^{-23}\times2^{-126}$
♦ double ≈ 4.9×10^{-324}			$2^{-52}\times2^{-1022}$
◈ Наиб. положит. денорм.	00…00	11…11	
♦ float ≈ 1.18×10^{-38}			$(1.0 - \varepsilon)\times2^{-126}$
♦ double ≈ 2.2×10^{-308}			$(1.0 - \varepsilon)\times2^{-1022}$
◈ Наим. положит. норм.	00…01	00…00	
♦ float			1.0×2^{-126}
♦ double			1.0×2^{-1022}
◈ Единица	01…11	00…00	1.0
◈ Наиб. положит. норм.			
♦ float ≈ 3.4×10^{38}			$(2.0 - \varepsilon)\times2^{127}$
♦ double ≈ 1.8×10^{308}			$(2.0 - \varepsilon)\times2^{1023}$