Семестр 2 (2018), занятие 1

План:

- беззнаковые целые числа;
- представление вещественных чисел;
- порядковые номера чисел с плавающей точкой;
- округление вещественных чисел;
- вычислительная погрешность;
- задание режимов округления;
- задачи;
- константы;
- преобразование строки в число.

1 Беззнаковые целые числа

Следующие типы данных

- `unsigned char`,
- `unsigned short`,
- `unsigned int`,
- `long unsigned int`,
- `long long unsigned int`

используются для работы с беззнаковыми целыми числами.

Применение операции `sizeof` к эти типам может вернуть следующие значения 1, 2, 4, 8, 8.

В стандартном файле `inttypes.h` определены также следующие типы

- `uint8_t`,
- `uint16_t`,
- `uint32_t`,
- `uint64_t`.

Распечатать битовое представление значения типа `long long unsigned int`.

Листинг 1: Функция `print`.

```
void printLLUI(long long unsigned int x) {
    int j;
    for(j = 63; j >= 0; j--)
        printf("%c ", (x & 1llu << j)
                          ? '1' : '0');
    printf("\n%llu\n", x);
}
```

Распечатать битовое представление значения типа `uint64_t`.

Листинг 2: Функция `print`.

```
#include <inttypes.h>

void printUint64(uint64_t x) {
    int j;
    for(j = 63; j >= 0; j--)
        printf("%c ", (x & UINT64_C(1) << j)
                          ? '1' : '0');
    printf("\n%"PRIu64"\n", x);
}
```

2 Представление вещественных чисел

Числа с плавающей точкой двойной точности, которым соответствует тип double в языке Си, представляются последовательностями длинной в 64 бита. Кодирующая последовательность имеет следующий формат.

$$s\, p_{10}\, p_9 \ldots p_1\, p_0\; m_1\, m_2 \ldots m_{51}\, m_{52} \qquad (1)$$

Старший бит s задает знак числа. Последовательность битов $p_{10} \ldots p_0$ определяют показатель степени. Будем интерпретировать эту последовательность как запись некоторого числа p в двоичной системе счисления. Наконец, последовательность битов $m_1 \ldots m_{52}$ кодирует мантиссу. Возможны следующие варианты.

Вариант 1. Число $p \neq 0$ и $p \neq 2047$. Это значит, что последовательность $p_{10} \ldots p_0$ не состоит целиком только из нулей или только из единиц. В этом случае (1) задает нормализованное число, которое вычисляется по следующей формуле

$$(-1)^s\; 1.m_1\, m_2 \ldots m_{51}\, m_{52}\; 2^{p-1023}. \qquad (2)$$

В (2) число $1.m_1\, m_2 \ldots m_{51}\, m_{52}$ записано в двоичной системе счисления.

Вариант 2. Число $p = 0$ и мантисса не состоит целиком из нулей. В этом случае (1) задает денормализованное число, которое вычисляется по следующей формуле

$$(-1)^s \; 0.m_1\, m_2 \ldots m_{51}\, m_{52} \; 2^{-1022}. \qquad (3)$$

В (3) число $0.m_1\, m_2 \ldots m_{51}\, m_{52}$ записано в двоичной системе счисления.

Вариант 3. Число $p = 0$ и мантисса состоит целиком из нулей. В этом случае (1) задает нулевые числа. При $s = 0$ (1) задает нуль, а при $s = 1$ задает так называемый отрицательный нуль.

Вариант 4. Число $p = 2047$ и мантисса состоит целиком из нулей. В этом случае (1) задает значение, которое называется бесконечностью. При $s = 0$ это значение называется положительной бесконечностью *Inf*, а при $s = 1$ называется отрицательной бесконечностью *-Inf*.

Вариант 5. Число $p = 2047$ и мантисса не состоит целиком из нулей. В этом случае (1) задает значение, которое называется «не число» *NaN*.

Обозначим через $\mathbb{F}$ множество всех чисел с плавающей точкой двойной точности (далее, просто числа с плавающей точкой), а через $\mathbb{R}$ – множество всех вещественных чисел. Множество $\mathbb{F}$ конечно и состоит из 2^{64} элементов. Пересечение множеств $\mathbb{F} \cap \mathbb{R}$ состоит из нуля, нормализованных и денормализованных чисел.

Последовательность битов (1) может кодировать как число с плавающей точкой $x \in \mathbb{F}$, так и беззнаковое целое число $n(x)$, которое будем называть *порядковым номером* x. Порядковые номера обладают следующими свойствами.

Для положительных чисел $x_1, x_2 \in \mathbb{F} \cap \mathbb{R}$ неравенство $x_1 < x_2$ выполняется тогда и только тогда, когда выполняется неравенство $n(x_1) < n(x_2)$. Если $n(x_2) = n(x_1) + 1$, то открытый интервал вещественных чисел (x_1, x_2) не содержит чисел с плавающей точкой.

Для отрицательных чисел $x_1, x_2 \in \mathbb{F} \cap \mathbb{R}$ неравенство $x_1 < x_2$ выполняется тогда и только тогда, когда выполняется неравенство $n(x_1) > n(x_2)$. Если $n(x_2) = n(x_1) - 1$, то открытый интервал вещественных чисел (x_1, x_2) не содержит чисел с плавающей точкой.

3 Порядковые номера чисел с плавающей точкой

Число с плавающей точкой и его порядковый номер кодируются одинаковой последовательностью битов. Поэтому последовательность из восьми байтов, хранящуюся в некоторой области памяти адресного пространства выполняющейся программы, можно интерпретировать и как бинарное представление числа с плавающей точкой, так и как бинарное представление беззнакового целого числа, которое будет являться порядковым номером этого числа с плавающей точкой.

В листинге 3 с помощью операций над указателями область памяти, выделенную под хранение переменной типа `double`, интерпретируется как область памяти, хранящую значение целочисленного типа `uint64_t`. Это значение сохраняется в переменную `i` и является порядковым номером числа 1.5.

Листинг 3: Пример использования указателей.

```
#include <inttypes.h>
...
double   d = 1.5;
uint64_t i = *(uint64_t *)&d;
```

В листинге 4 продемонстрирован другой подход. Определяется новый тип данных `real_t` в виде объединения с двумя полями типа `double` и типа `uint64_t`. Со значением типа `real_t` можно одновременно работать и как с числом с плавающей точкой (через поле `d`), и как с его порядковым номером (через поле `i`). Порядковым номером числа с плавающей точкой 1.5 является целое число 4609434218613702656. Поэтому переменные `r1` и `r2` будут иметь одинаковое значение.

Листинг 4: Пример использования объединений.

```
#include <inttypes.h>
...
typedef union {
   double   d;
   uint64_t i;
} real_t;
...
real_t r1, r2;
r1.d = 1.5;
r2.i = 4609434218613702656;
```

Задача формирования числа с плавающей точкой, соответствующего заданной кодирующей последовательности битов, может быть решена двумя способами. Первый способ базируется на использовании битовых операций и введенного типа данных `real_t`. Второй способ базируется на использовании шестнадцатеричных литералов. Рассмотрим каждый из них.

В качестве примера возьмем максимальное нормализованное число N_{max} и минимальное положительное денормализованное число D_{min}. Число N_{max} кодируется последовательностью битов

0111111111101111111111111111111111111111111111111111111111111111,

а число D_{min} кодируется последовательностью битов

0000000000000000000000000000000000000000000000000000000000000001.

Листинг 5 содержит фрагмент программы, в результате выполнения которого значением выражения `r1.d` будет число N_{max}, а значением выражения `r2.d` будет число D_{min}.

Листинг 5: Формирование чисел N_{max} и D_{min}.

```
real_t r1, r2;
r1.i = ~( UINT64_C(1) << 63
        | UINT64_C(1) << 52);
r2.i = 1;
```

Константы типа `double` могут быть заданы с помощью шестнадцатеричных литералов. В листинге 6 с их помощью задаются числа N_{max} и D_{min}. Отметим, что формат шестнадцатеричного литерала хорошо согласуется с представлениями (2) и (3). Отличий всего два. Для задания мантиссы используется шестнадцатеричная система счисления вместо двоичной системы. Показатель степени числа 2 задан явно.

Листинг 6: Пример использования шестнадцатеричных литералов.

```
double Nmax = 0x1.fffffffffffffp+1023,
       Dmin = 0x0.0000000000001p-1022;
```

4 Округление вещественных чисел

Не каждое вещественное число может быть точно представлено в виде числа с плавающей точкой. В этом случае, вместо вещественного числа $x \in \mathbb{R}$ используется некоторое приближенное к нему значение $R(x) \in \mathbb{F}$. Функция R называется *функцией округления*.

Необходимость в округлении возникает в следующих ситуациях. В текстах программ вещественные константы обычно задаются с помощью десятеричных литералов. В результате компилятору во время трансляции программы приходится округлять значение каждого десятеричного литерала, когда это значение не может быть точно представлено в виде числа с плавающей точкой. Примером такого литерала может служить 0.1. Отметим, что такая необходимость не возникает при использовании шестнадцатеричных литералов, однако, они не так широко используются.

Потребность в округлении также возникает при выполнении арифметических операций. Например, если $x_1, x_2 \in \mathbb{R} \cap \mathbb{F}$, то результатом вычисления выражения $x_1 + x_2$ будет значение $R(x_1 + x_2) \in \mathbb{F}$.

Рассмотрим общие для всех функций округления свойства. Очевидно, что должно выполняться равенство $R(x) = x$ для всякого $x \in \mathbb{F}$. Точно также очевидно, что $R(y) \neq y$ для всякого $y \in \mathbb{R} \setminus \mathbb{F}$. Несмотря на распространенность ситуации, когда $R(y) \neq y$, она трактуется как ошибка в операции над числами с плавающими точками (далее, ошибками) *неточный результат* (*inexact*) Выделяют три случая, схематично изображенные на рис. 1.

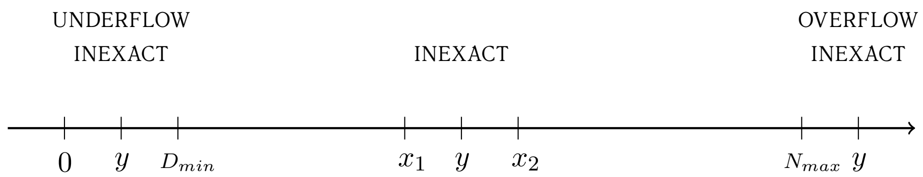

Рис. 1: Округление и ошибки. Три случая.

Случай 1, $D_{min} < y < N_{max}$. В этом случае существуют два подряд идущих числа с плавающей точкой $x_1, x_2 \in \mathbb{F}$ такие, что $y \in (x_1, x_2)$. В качестве значения $R(y)$ выбирается один из концов этого интервала.

Случай 2, $y > N_{max}$. В качестве значения

$R(y)$ будет выступать либо максимальное нормализованное число N_{max}, либо специальное значение бесконечность *Inf*. В некоторых случаях эта ситуация трактуется как ошибка *переполнение* (*overflow*).

Случай 3, $0 < y < D_{min}$. В качестве значения $R(y)$ будет выступать либо нуль, либо минимальное положительное денормализованное число D_{min}. В некоторых случаях эта ситуация трактуется как ошибка *антипереполнение* (*underflow*).

Аналогично можно выделить и рассмотреть три случая для отрицательного числа y.

К числу стандартных ошибок также относят *деление на нуль* (*divbyzero*) и *недействительная операция* (*invalid*). При возникновении ошибки деления на нуль результатом арифметической операции становится специальное значение бесконечность *Inf*. Прибавление к значению Inf числа снова порождает значение *Inf*, и это выглядит вполне логично. Однако для ряда случаев трудно дать разумную интерпретацию результату арифметической операции. Например, *Inf* − *Inf*, $0 * Inf$, $0/0$, *Inf*/*Inf*. В этих случаях результат арифметической операции полагают равным специальному значению *NaN* и трактуют эту ситуацию как ошибка недействительная операция.

Имеется четыре основных метода округления RN, RU, RD, RZ. Результаты их использования для случая, когда округляется число из интервала (D_{min}, N_{max}) или интервала $(-N_{max}, -D_{min})$, схематично изображены на рис. 2. На рисунке выделены два подряд идущих числа с плавающей точкой $x_1, x_2 \in \mathbb{F}$ ($x_1 > 0$), а также три вещественных числа $y_1, y_2, y_3 \in (x_1, x_2)$. При этом y_2 – является серединой интервала (x_1, x_2), и выполняются строгие неравенства $y_1 < y_2 < y_3$.

Метод RN осуществляет округление в сторону ближайшего числа с плавающей точкой. Поэтому $RN(y_1) = x_1$ и $RN(y_3) = x_2$. Вещественное число y_2 находится ровно посередине интервала (x_1, x_2) и равноудалено от его концов. В этом случае рассматривают порядковые номера концов интервала и выбирают конец интервала с четным номером. На рис. 2 изображена ситуация, когда $n(x_1)$ четно, поэтому $RN(y_2) = x_1$.

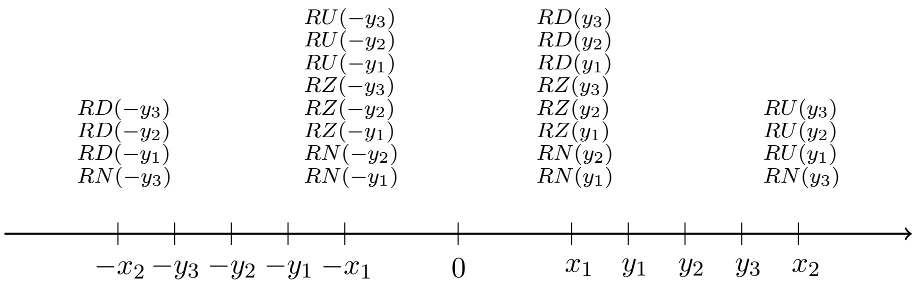

Рис. 2: Стандартные способы округления.

Метод RU осуществляет округление в сторону положительной бесконечности. Всегда выбирается правый конец интервала, поэтому $RU(y_1) = RU(y_2) = RU(y_3) = x_2$. Метод RD осуществляет округление в сторону отрицательной бесконечности. Всегда выбирается левый конец интервала, поэтому $RD(y_1) = RD(y_2) = RD(y_3) = x_2$. Наконец метод RZ осуществляет округление в сторону нуля. Всегда выбирается конец ближайший к нулю, поэтому, например, $RZ(y_2) = x_1$ и $RZ(-y_2) = -x_1$.

При закреплении данного материала на практических занятиях можно в качестве x_1 выбрать число 2^{56}, при этом окажется, что $x_2 - x_1 = 16$. Серединой интервала (x_1, x_2) будет число $y_2 = x_1 + 8$. В качестве y_1 и y_3 можно взять следующие числа. Пусть числа $a, b \in \mathbb{F}$ такие, что $n(a) = n(8) - 1$ и $n(b) = n(8) + 1$, тогда положим $y_1 = x_1 + a$ и $y_3 = x_1 + b$.

Рекомендуется написать программу, которая в разных режимах округления вычисляет y_1, y_2, y_3 и сравнивает их с x_1, x_2.

5 Вычислительная погрешность

С понятием округления тесно связано понятие вычислительной погрешности. В силу необходимости выполнения округлений во время выполнения последовательности арифметических операций получается не точный результат x, а его некоторое приближение x^*. Если имеет место неравенство $|x - x^*| \leqslant \delta$, то величину δ называют *абсолютной погрешностью* приближения.

Существуют теоретические оценки для абсолютных погрешностей результатов арифметических операций. Например, при вычислении суммы n чисел $x_1 + x_2 + \ldots + x_n$ погрешность будет иметь следующий вид

$$E_1|x_1| + E_2|x_2| + \ldots + E_n|x_n|, \qquad (4)$$

где $E_1 > E_2 > \ldots > E_n > 0$. Оценка (4) будет минимальной, если осуществлять суммирование в порядке возрастания абсолютных величин слагаемых.

6 Задание режимов округления

С помощью функции `fesetround` можно установить режим округления. Значениями входного параметра этой функции могут выступать константы `FE_TONEAREST`, `FE_UPWARD`, `FE_DOWNWARD`, `FE_TOWARDZERO`. В случае успеха функция возвращает нулевое значение.

Ниже приведен фрагмент программы, позволяющей установить режим округления к нулю.

```
#include <fenv.h>
 ...
fesetround(FE_TOWARDZERO);
```

7 Задачи

0. *Машинной точностью* называется максимальное положительное значение типа **double**, для которого выполняется равенство

```
1. + e = 1.
```

Вычислить (приближенно) машинную точность.

1. Реализовать функцию

void `print(`**double** `x)`,

которая печатает двоичную последовательность, кодирующую x, порядковый номер x, число x.

Например, в результате вызова `print(15.625)` должно быть напечатано

```
0 10000000010 1111010...0
4624985711076966400
1.5625000000000000e+01
```

Проверить функцию `print` на значениях 15.625, -15.625, e (`M_E`), π (`M_PI`).

2. Реализовать функцию

double `abs_(`**double** `x)`,

возвращающую абсолютную величину x. Запрещается использовать условные операторы и условные операции.

Проверить функцию `abs_` на значениях 15.625, -15.625, e, $-e$, π, $-\pi$.

3. Сформировать максимальное нормализованное число, минимальное положительное нормализованное число, `+Inf`, `NaN`. Распечатать эти числа с помощью функции `print`.

```
0 11111111110 1...11
9218868437227405311
1.7976931348623157e+308

0 00000000001 0...00
4503599627370496
2.2250738585072014e-308

0 11111111111 0...00
9218868437227405312
inf

0 11111111111 0...01
9218868437227405313
nan
```

4. Вычислить два нормализованных числа x_1 и x_2 таких, что x_1 – степень числа 2, $n(x_2) = n(x_1) + 1$ и $x_2 - x_1 = 16$. Распечатать эти числа с помощью функции `print`.

```
4859383997932765184
7.2057594037927936e+16

4859383997932765185
```

```
7.2057594037927952e+16
```

5. Пусть x_1 – число из предыдущей задачи, а y_1 и y_2 – нормализованные числа такие, что $n(y_1) = n(8) - 1$ и $n(y_2) = n(8) + 1$. Вычислить $x_1 + y_1$, $x_1 + 8$, $x_1 + y_2$ в режимах округления `FE_TONEAREST`, `FE_UPWARD`, `FE_TOWARDZERO`.

Распечатать вычисленные значения с помощью функции `print`.

Листинг 7: Функция `print`.

```
void print(double x) {
    real_t r;
    int    j;

    for(j = 63, r.d = x; j >= 0; j --) {
        printf("%c", (r.i & UINT64_C(1) << j)
                     ? '1'
                     : '0');
        if(j == 63 || j == 52)
            printf(" ");
    }
    printf("\n");

    printf("%"PRIu64"\n", r.i);
    printf("%.15e\n", x);
}
```

8 Константы

Число 1.5.

```
0 01111111111 1000000000000000000000000000000000000000000000000000
4609434218613702656
1.5000000000000000e+00

+ 1.1000000000000000000000000000000000000000000000000000 2^0
```

Число -1.5.

```
1 01111111111 1000000000000000000000000000000000000000000000000000
13832806255468478464
-1.5000000000000000e+00

- 1.1000000000000000000000000000000000000000000000000000 2^0
```

Число 15.625.

```
0 10000000010 1111010000000000000000000000000000000000000000000000
4624985711076966400
1.5625000000000000e+01

+ 1.1111010000000000000000000000000000000000000000000000 2^3
```

Число -15.625.

```
1 10000000010 1111010000000000000000000000000000000000000000000000
13848357747931742208
-1.5625000000000000e+01

- 1.1111010000000000000000000000000000000000000000000000 2^3
```

Число π.

```
0 10000000000 1001001000011111101101010100010001000010110100011000
4614256656552045848
3.1415926535897931e+00

+ 1.1001001000011111101101010100010001000010110100011000 2^1
```

Число e.

```
0 10000000000 0101101111110000101010001011000101000101011101101001
4613303445314885481
2.7182818284590451e+00

+ 1.0101101111110000101010001011000101000101011101101001 2^1
```

Максимальное положительное нормализованное число.

```
0 11111111110 1111111111111111111111111111111111111111111111111111
9218868437227405311
1.7976931348623157e+308
```

```
+ 1.1111111111111111111111111111111111111111111111111111 2^1023
```

Минимальное положительное нормализованное число.

```
0 00000000001 0000000000000000000000000000000000000000000000000000
4503599627370496
2.2250738585072014e-308

+ 1.0000000000000000000000000000000000000000000000000000 2^-1022
```

Максимальное положительное денормализованное число.

```
0 00000000000 1111111111111111111111111111111111111111111111111111
4503599627370495
2.2250738585072009e-308

+ 0.1111111111111111111111111111111111111111111111111111 2^-1022
```

Минимальное положительное денормализованное число.

```
0 00000000000 0000000000000000000000000000000000000000000000000001
1
4.9406564584124654e-324

+ 0.0000000000000000000000000000000000000000000000000001 2^-1022
```

Значение `+Inf`.

```
0 11111111111 0000000000000000000000000000000000000000000000000000
9218868437227405312
inf

+ 1.0000000000000000000000000000000000000000000000000000 2^1024
```

Значение `NaN`.

```
0 11111111111 0000000000000000000000000000000000000000000000000001
9218868437227405313
nan

+ 1.0000000000000000000000000000000000000000000000000001 2^1024
```

9 Преобразование строки в число

```
typedef struct {
    union {
        int    i;
        double d;
    }   val;
    int hasVal;
} maybe_t;


#include <stdlib.h>
#include <limits.h>
#include <errno.h>

maybe_t toDouble(const char *str) {
    double  d;
    char   *e;

    errno = 0;
    d = strtod(str, &e);

    if(!errno && *e == '\0')
        return (maybe_t) {
            .val    = { .d = d },
            .hasVal = 1 };
    else
        return (maybe_t) {
            .val = { .i = 0 },
            .hasVal = 0 };
}


maybe_t toInt(const char *str) {
    long  l;
    char *e;

    errno = 0;
    l = strtol(str, &e, 10);

    if(!errno && *e == '\0')
        if(INT_MIN <= l && l <= INT_MAX)
            return (maybe_t) {
                .val    = { .i = (int)l },
                .hasVal = 1 };

    return (maybe_t) {
        .val    = { .i = 0 },
        .hasVal = 0 };
}
```