МИНОБРНАУКИ РОССИИ

–––––––——————————–––––––

Санкт-Петербургский государственный  
электротехнический университет «ЛЭТИ»

————————————————————

**Лабораторная работа №1 по теме**

**«Исследование внутреннего представления различных форматов данных»**

Санкт-Петербург

Издательство СПБГЭТУ «ЛЭТИ»

2024

**Цель работы**

Знакомство с внутренним представлением различных типов данных, используемых компьютером при их обработке.

**Задание (вариант 5)**

1. Разработать алгоритм ввода с клавиатуры int и long double число и показать на экране его внутреннее представление в двоичной системе счисления.

2. Написать и отладить программу на языке С, реализующую разработан-ный алгоритм. Программа должна

- иметь дружественный интерфейс

- выводить на экран информативное сообщение при вводе некорректных дан-ных

- предложить повторный ввод пока не будут введены корректные данные

3. Поменять местами заданные пользователем группы рядом стоящих бит, номера старших разрядов этих групп и количество бит в группе, вводится с клавиатуры.

**Описание решения**

Рассмотрим два различных решения задачи – для int и для long double. Так как int – целое число, то оно поддерживает битовые операции, а потому можно:

1. Создать битовые маски для групп
2. Обнулить биты в исходном числе
3. Использовать побитовое ИЛИ между исходным числом и группами, чтобы поменять их местами

Теперь подробнее о создании получении группы:

Так как пользователь вводит позицию слева направо, то необходимо сдивнуть исходное число на (sizeOfInt - pos - len + 1), чтобы отсечь ненужные биты, запишем результат этой операции в переменную result. Затем создадим маску, содержащую нули во всех полях, кроме младших разрядов длиной len. А после использовать побитовое И между result и маской.

Чтобы положить наш result в новую позицию нужно выполнить следующий алгоритм:

1. Сначала определим сдвиг delta = (sizeOfInt - pos - len + 1).
2. Затем нам надо обнулить все биты, соответствующие группе, в исходном числе.
3. Затем мы делаем побитовое ИЛИ между исходным числом и группой, сдвинутой на delta.

Данные операции выполняются дважды – для обеих групп.

В случае с переменной формата long double решение задачи меняется. Дробные числа не поддерживают побитовые сдвиги – это связано с тем, как они хранятся в памяти. Рассмотрим на примере float и double.

float

|  |  |  |
| --- | --- | --- |
| S | P | M |

31 30 23 22 0

double

|  |  |  |
| --- | --- | --- |
| S | P | M |

63 62 52 51 0

Дробные числа в памяти хранятся в виде трёх частей:

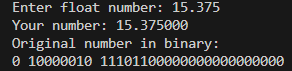
1. Знаковый бит (0 для положительных и 1 для отрицательных)
2. Порядок (степень, в которую будет вовзодиться двойка)
3. Мантисса (дробное число от 1 до 2, но в случае float и double хранится только дробная часть, т к единица хранится всегда, так что для более эффективного использования памяти единицу опускают).

В конце идёт перемножение S \* 2P \* 1.M

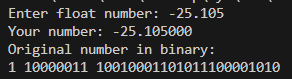
Также следует отметить, что порядок хранится со смещение на 127 бит в случае с float и 1023 бита для double. Это требуется для записи отрицательных степеней двойки.

Рассмотрим примеры (для float):

1. Число 15.375

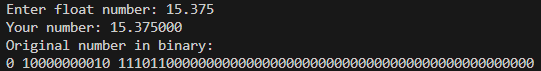


1. Число -25.105

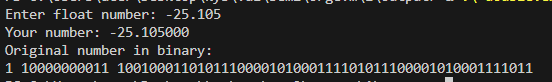


Рассмотрим примеры (для double):

1. Число 15.375



1. Число -25.105



Теперь к алгоритму для long double:

Так как long double имеет размер больший, чем любое из целых чисел, то для решения задачи используется union из long double и массива unsigned char, поскольку размер такой переменной – 1 байт.

После ввода long double и указания расположения групп, необходимо создать два массива unsigned char такого же размера, что и в union, занести в них биты, соответствующие группам, после обнулить биты групп в исходном массиве и перезаписать их, используя ранее созданные два массива.

Суть операций такая же как и в случае с переменной int, но теперь нужно работать с массивами.

Основная идея заключается в использовании цикла от 0 до len, в котором вычисляется byteIndex1, byteIndex2 и bitIndex1, bitIndex2 на основе позиций и текущего значения переменной цикла. Так вычисляется нужная позиция внутри массива, что позволяет осуществлять запись. Для «сдвига» наших групп мы вычисляем shift как разницу между бит индексами, после чего смотрим на его знак: если он отрицательный, то сдвигаем первую группу вправо на (-shift) а вторую влево на (-shift), иначе необходимо сдвинуть первую группу влево на shift, а вторую вправо на shift.

**Исходный код программы для обработки long double и int**

#include <stdio.h>

#define ldBytes sizeof(long double)

union Data {

long double number;

unsigned char arr[ldBytes];

};

// int isLittleEndian() {

// unsigned int x = 1;

// return \*((unsigned char\*)&x) == 1;

// }

void printBinaryChar(unsigned char n) {

int i, bit;

for (i = 7; i >= 0; i--) {

bit = (n >> i) & 1;

printf("%d", bit);

}

}

void printBits(unsigned char arr[]) {

int i;

for (i = ldBytes - 1; i >= 0; i--) {

printBinaryChar(arr[i]);

printf(" ");

}

// if (isLittleEndian()) {

// for (i = ldBytes - 1; i >= 0; i--) {

// printBinaryChar(arr[i]);

// printf(" ");

// }

// } else {

// for (i = 0; i < ldBytes; i++) {

// printBinaryChar(arr[i]);

// printf(" ");

// }

// }

printf("\n");

}

void swapBitGroups(union Data\* data) {

int pos1, len, pos2, i, j, bitIndex, byteIndex1, byteIndex2, bitIndex1, bitIndex2, shift;

unsigned char group1[ldBytes] = {0}, group2[ldBytes] = {0};

printf("Enter the position of the fisrt bit of the first group (from left to right): ");

scanf("%d", &pos1);

printf("Enter the position of the first bit of the second group (from left to right): ");

scanf("%d", &pos2);

printf("Enter the number of bits in the groups: ");

scanf("%d", &len);

if (pos1 < 0 || pos1 >= ldBytes \* 8 || len <= 0 || (pos1 + len) > ldBytes \* 8) {

printf("Error: invalid parameters for the first group.\n");

return;

}

if (pos2 < 0 || pos2 >= ldBytes \* 8 || len <= 0 || (pos2 + len) > ldBytes \* 8) {

printf("Error: invalid parameters for the second group.\n");

return;

}

if ((pos2 + len > ldBytes \* 8) || (pos1 + len > ldBytes \* 8)) {

printf("Error: groups are too long to rearrange.\n");

return;

}

// for (i = 0; i < ldBytes; i++) {

// for (j = 0; j < 8; j++) {

// bitIndex = i \* 8 + j;

// if ((bitIndex >= pos1) && (bitIndex < (pos1 + len))) {

// if (data->arr[ldBytes - i - 1] & (1 << (7 - j))) {

// group1[ldBytes - i - 1] |= (1 << (7 - j));

// }

// }

// if ((bitIndex >= pos2) && (bitIndex < (pos2 + len))) {

// if (data->arr[ldBytes - i - 1] & (1 << (7 - j))) {

// group2[ldBytes - i - 1] |= (1 << (7 - j));

// }

// }

// }

// }

for (i = 0; i < len; i++) {

byteIndex1 = ldBytes - (i + pos1) / 8 - 1;

byteIndex2 = ldBytes - (i + pos2) / 8 - 1;

bitIndex1 = 7 - ((i + pos1) % 8);

bitIndex2 = 7 - ((i + pos2) % 8);

if (data->arr[byteIndex1] & (1 << bitIndex1)) {

group1[byteIndex1] |= (1 << bitIndex1);

}

if (data->arr[byteIndex2] & (1 << bitIndex2)) {

group2[byteIndex2] |= (1 << bitIndex2);

}

}

// for (i = 0; i < ldBytes; i++) {

// for (j = 0; j < 8; j++) {

// bitIndex = i \* 8 + j;

// if ((bitIndex >= pos1) && (bitIndex < (pos1 + len))) {

// data->arr[ldBytes - i - 1] &= ~(1 << (7 - j));

// }

// if ((bitIndex >= pos2) && (bitIndex < (pos2 + len))) {

// data->arr[ldBytes - i - 1] &= ~(1 << (7 - j));

// }

// }

// }

for (i = 0; i < len; i++) {

byteIndex1 = ldBytes - (i + pos1) / 8 - 1;

byteIndex2 = ldBytes - (i + pos2) / 8 - 1;

bitIndex1 = 7 - ((i + pos1) % 8);

bitIndex2 = 7 - ((i + pos2) % 8);

data->arr[byteIndex1] &= ~(1 << bitIndex1);

data->arr[byteIndex2] &= ~(1 << bitIndex2);

}

printf("group1: \n");

printBits(group1);

printf("group2: \n");

printBits(group2);

for (i = 0; i < len; i++) {

byteIndex1 = ldBytes - (i + pos1) / 8 - 1;

byteIndex2 = ldBytes - (i + pos2) / 8 - 1;

bitIndex1 = ((i + pos1) % 8);

bitIndex2 = ((i + pos2) % 8);

shift = bitIndex1 - bitIndex2;

if (shift < 0) {

data->arr[byteIndex1] |= group2[byteIndex2] << (-shift);

data->arr[byteIndex2] |= group1[byteIndex1] >> (-shift);

} else {

data->arr[byteIndex1] |= group2[byteIndex2] >> (shift);

data->arr[byteIndex2] |= group1[byteIndex1] << (shift);

}

}

}

int main() {

union Data data;

printf("Enter long double number: ");

scanf("%Lf", &data.number);

printf("Your number: %Lf\n", data.number);

printf("Original number in binary:\n");

printBits(data.arr);

swapBitGroups(&data);

printf("After swapping:\n");

printBits(data.arr);

printf("Your number: %Lf\n", data.number);

return 0;

}

**Исходный код программы для обработки double**

#include <stdio.h>

const int sizeofLong = sizeof(long long) \* 8;

union Data {

double d;

long long l;

};

void printBinary(long long n) {

for (int i = sizeofLong - 1; i >= 0; i--) {

int bit = (n >> i) & 1;

printf("%d", bit);

if (i == 63 || i == 52) {

printf(" ");

}

}

printf("\n");

}

int main() {

union Data data;

printf("Enter float number: ");

scanf("%lf", &data.d);

printf("Your number: %lf\n", data.d);

printf("Original number in binary:\n");

printBinary(data.l);

return 0;

}

**Исходный код программы для обработки float**

#include <stdio.h>

const int sizeofLong = sizeof(int) \* 8;

union Data {

float f;

int i;

};

void printBinary(int n) {

for (int i = sizeofLong - 1; i >= 0; i--) {

int bit = (n >> i) & 1;

printf("%d", bit);

if (i == 31 || i == 23) {

printf(" ");

}

}

printf("\n");

}

int main() {

union Data data;

printf("Enter float number: ");

scanf("%f", &data.f);

printf("Your number: %f\n", data.f);

printf("Original number in binary:\n");

printBinary(data.i);

return 0;

}

**Выводы**

В ходе выполнения работы мы изучили, как компьютер хранит и обрабатывает различные типы данных, а также подробно рассмотрели двоичное представление целых и дробных чисел и операции сдвига битов. Мы узнали и применили следующие ключевые моменты:

Мы изучили, как целые числа хранятся в памяти компьютера в виде двоичных кодов. Для этого было реализовано отображение чисел в двоичном виде, что позволило визуализировать, как каждый бит числа используется для представления данных.

Рассмотрели особенности хранения как положительных, так и отрицательных чисел с использованием дополнительного кода. Этот метод позволяет эффективно представлять отрицательные числа и выполнять арифметические операции.

Мы рассмотрели три основных метода представления чисел со знаком: прямой код, обратный код и дополнительный код. Узнали, что прямой и обратный коды имеют недостатки (например, два представления для нуля), и что дополнительный код широко используется благодаря удобству выполнения арифметических операций.

Мы научились работать с побитовыми сдвигами чисел, как влево, так и вправо. Рассмотрели разницу между арифметическим и логическим сдвигом, особенно в контексте обработки отрицательных чисел, где важно сохранять или изменять знаковый бит.

Применили эти знания для обработки групп битов, извлекая их из числа и меняя их местами.

Также в процессе работы мы изучили темы представления чисел с плавающей запятой, где изучили структуру числа: знак, экспонента и мантисса. Это дало понимание, как числа с плавающей запятой хранятся в двоичной системе.

Был разработан и реализован алгоритм, который позволяет пользователю задавать две группы битов, а затем менять их местами. Для этого мы научились извлекать группы битов, обрабатывать их и устанавливать на новые позиции.

Таким образом, мы научились эффективно работать с двоичными представлениями данных, узнали, как компьютер хранит целые числа и числа с плавающей запятой, и реализовали операции сдвига и перестановки битовых групп с учётом внутренней структуры данных.