Лабораторная работа №6. Битовые операции. Аргументы командной строки. Сравнение вещественных чисел.

Исследование применимости алгоритмов для сравнения вещественных чисел на равенство. Применение алгоритмов обработки некорректного пользовательского ввода и алгоритмов обработки ложных ожиданий, ввод данных через аргументы командной строки (терминал), действия с разрядами целых беззнаковых чисел.

Теоретическая часть

Аргументы командной строки

Исполняемые программы могут запускаться в командной строке через вызов. Например, для запуска исполняемого файла MyProgram, который находится в корневом каталоге диска C в ОС Windows, вам нужно ввести:

```
C:\>MyProgram
```

Чтобы передать аргументы командной строки в MyProgram, вам нужно будет их просто перечислить после имени исполняемого файла:

```
C:\>MyProgram SomeContent.txt
```

Теперь, при запуске MyProgram, SomeContent.txt будет предоставлен в качестве аргумента командной строки. Программа может иметь несколько аргументов командной строки, разделённых пробелами:

```
C:\>MyProgram SomeContent.txt SomeOtherContent.txt
```

Использование аргументов командной строки

```
int main(int argc, char *argv[])
```

- argc — содержащий количество аргументов, переданных в программу (минимум один, так как первым аргументом всегда является имя самой программы).
- argv — фактические значения аргументов, массив строк C-style, количество элементов этого массива равно argc.

```
// Программа MyArguments выводит значения всех аргументов командной строки
#include <iostream>
int main(int argc, char *argv[])
{
    std::cout << "There are " << argc << " arguments:\n";
    // Перебираем каждый аргумент и выводим его порядковый номер и значение
    for (int count=0; count < argc; ++count)
        std::cout << count << " " << argv[count] << '\n';
    return 0;
}
```

Вывод будет следующим:

```
There are 3 arguments:
0 C:\MyArguments
1 SomeContent.txt
2 123
```

Обработка числовых аргументов

Аргументы командной строки всегда передаются в качестве строк, даже если предоставленное значение является числовым. Чтобы использовать аргумент командной строки в виде числа, вам нужно будет конвертировать его из строки в число.

```
#include <iostream>
#include <string>
#include <sstream> // для std::stringstream
#include <cstdlib> // для exit()

int main(int argc, char *argv[])
{
	if (argc <= 1)
	{
// В некоторых операционных системах argv[0] может быть просто пустой строкой, без имени программы
// Обрабатываем случай, когда argv[0] может быть пустым или не пустым
		if (argv[0])
			std::cout << "Usage: " << argv[0] << " <number>" << '\n';
		else
			std::cout << "Usage: <program name> <number>" << '\n';
		exit(1);
	}
	std::stringstream convert(argv[1]); // создаем переменную stringstream с именем convert, инициализируя её значением argv[1]
	int myint;
	if (!(convert >> myint)) // выполняем конвертацию
		myint = 0; // если конвертация терпит неудачу, то присваиваем myint значение по умолчанию
	std::cout << "Got integer: " << myint << '\n';
	return 0;
}
```

Если мы запустим эту программу с аргументом командной строки 843, то результатом будет:

```
Got integer: 843
```

Алгоритм сравнения вещественных чисел

Сравнение значений типа с плавающей точкой с помощью любого из этих операторов — дело опасное. Почему? Из-за тех самых небольших ошибок округления, которые могут привести к неожиданным результатам. Например:

```
#include <iostream>

int main()
{
    double d1(100 - 99.99); // должно быть 0.01
    double d2(10 - 9.99); // должно быть 0.01

    if (d1 == d2)
        std::cout << "d1 == d2" << "\n";
```

```
    else if (d1 > d2)
        std::cout << "d1 > d2" << "\n";
    else if (d1 < d2)
        std::cout << "d1 < d2" << "\n";

    return 0;
}
```

Вот так раз:

d1 > d2

В вышеприведенной программе, d1 = 0.0100000000000005116, а d2 = 0.0099999999999997868.
Часто пытаются использовать подобные функции определения равенства чисел:

```
#include <cmath> // для функции fabs()

bool isAlmostEqual(double a, double b, double epsilon)
{
    /* Если разница между a и b меньше значения эпсилона, то тогда a и b -
"достаточно близки" */
    return fabs(a - b) <= epsilon;
}
```

Хоть это и рабочий вариант, но он не идеален. Эпсилон 0.00001 подходит для чисел около 1.0, но будет слишком большим для чисел типа 0.0000001 и слишком малым для чисел типа 10000. Это означает, что каждый раз, при вызове функции, нам нужно будет выбирать наиболее соответствующий входным данным функции эпсилон. Дональд Кнут, «Искусство программирования, том 2: Получисленные алгоритмы» (1968):

```
#include <cmath> // для функции fabs()

// Возвращаем true, если разница между a и b в пределах процента эпсилона
bool approximatelyEqual(double a, double b, double epsilon)
{
    return fabs(a - b) <= ( (fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}
```

Здесь, вместо использования эпсилона как абсолютного числа, мы используем его как умножитель, чтобы подстроиться под входные данные. Но и функция approximatelyEqual() тоже не идеальна, особенно, когда дело доходит до чисел, близких к нулю:

```
#include <iostream>
#include <cmath> // для функции fabs()

// Возвращаем true, если разница между a и b в пределах процента эпсилона
bool approximatelyEqual(double a, double b, double epsilon)
{
      return fabs(a - b) <= ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}

int main()
{
      /* Значение a очень близкое к 1.0, но, из-за ошибок округления, чуть
меньше 1.0 */
      double a = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1;
```

```
	// Во-первых, давайте сравним значение a (почти 1.0) с 1.0
	std::cout << approximatelyEqual(a, 1.0, 1e-8) << "\n";

	// Во-вторых, давайте сравним значение a - 1.0 (почти 0.0) с 0.0
	std::cout << approximatelyEqual(a - 1.0, 0.0, 1e-8) << "\n";
}
```

Возможно, вы удивитесь, но результат:

```
1
0
```

Второй вызов не сработал так, как ожидалось. Математика просто ломается, когда дело доходит до нулей.
Но и этого можно избежать, используя как абсолютный эпсилон (то, что мы делали в первом способе), так и относительный (способ Кнута) вместе:

```
/* Возвращаем true, если разница между a и b меньше absEpsilon или в пределах
relEpsilon */
bool approximatelyEqualAbsRel(double a, double b, double absEpsilon, double
relEpsilon)
{
    /* Проверяем числа на их близость - это нужно в тех случаях, когда
сравниваемые числа являются нулевыми или "около нуля" */
    double diff = fabs(a - b);
    if (diff <= absEpsilon)
        return true;

    // В противном случае, возвращаемся к алгоритму Кнута
    return diff <= ( (fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * relEpsilon);
}
```

Здесь мы добавили новый параметр — `absEpsilon`. Сначала мы сравнивае `a` и `b` с `absEpsilon`, который должен быть задан как очень маленькое число (например, 1e-12). Таким образом, мы решаем случаи, когда `a и b` — нулевые значения или близки к нулю. Если это не так, то мы возвращаемся к алгоритму Кнута.
Протестируем:

```
#include <iostream>
#include <cmath> // для функции fabs()

// Возвращаем true, если разница между a и b в пределах процента эпсилона
bool approximatelyEqual(double a, double b, double epsilon)
{
	return fabs(a - b) <= ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}

// Возвращаем true, если разница между a и b меньше absEpsilon
// или в пределах relEpsilon
bool approximatelyEqualAbsRel(double a, double b, double absEpsilon, double
relEpsilon)
{
	// Проверяем числа на их близость - это нужно в случаях,
	// когда сравниваемые числа являются нулевыми или около нуля
	double diff = fabs(a - b);
	if (diff <= absEpsilon)
		return true;

	// В противном случае, возвращаемся к алгоритму Кнута
	return diff <= ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * relEpsilon);
```

```
}

int main()
{
      // Значение a очень близко к 1.0, но, из-за ошибок округления,
      // чуть меньше 1.0
      double a = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1;

// сравниваем "почти 1.0" с 1.0
      std::cout << approximatelyEqual(a, 1.0, 1e-8) << "\n";
// сравниваем "почти 0.0" с 0.0
      std::cout << approximatelyEqual(a - 1.0, 0.0, 1e-8) << "\n";
// сравниваем "почти 0.0" с 0.0
      std::cout << approximatelyEqualAbsRel(a - 1.0, 0.0, 1e-12, 1e-8) << "\n";
}
```

Результат:

```
1
0
1
```

С удачно подобранным absEpsilon, функция approximatelyEqualAbsRel() обрабатывает близкие к нулю и нулевые значения корректно.

Алгоритмы использующие битовые операции

Битовые маски

Включение, выключение, переключение или запрашивание сразу нескольких бит можно осуществить в одной битовой операции. Когда мы соединяем отдельные биты вместе, в целях их модификации как группы, то это называется битовой маской.
Рассмотрим пример. В следующей программе мы просим пользователя ввести число. Затем, используя битовую маску, мы сохраняем только последние 4 бита, значения которых и выводим в консоль:

```
#include <iostream>

int main()
{
    const unsigned int lowMask = 0xF; /* битовая маска для хранения последних 4-
х бит (шестнадцатеричный литерал для 0000 0000 0000 1111) */

    std::cout << "Enter an integer: ";
    int num;
    std::cin >> num;

    num &= lowMask; // удаляем первые биты, оставляя последние 4

    std::cout << "The 4 low bits have value: " << num << '\n';

    return 0;
}
```

Результат выполнения программы:

```
Enter an integer: 151
The 4 low bits have value: 7
```

151 в десятичной системе равно 1001 0111 в двоичной. lowMask — это 0000 1111 в 8-битной двоичной системе. 1001 0111 & 0000 1111 = 0000 0111, что равно десятичному 7.

Пример с RGBA

Цветные дисплейные устройства, такие как телевизоры и мониторы, состоят из миллионов пикселей, каждый из которых может отображать точку цвета. Точка цвета состоит из 3-х пучков: один красный, один зелёный и один синий (сокр. «RGB» от англ. «Red, Green, Blue»). Изменяя их интенсивность, можно воссоздать любой цвет. Количество цветов R, G и B в одном пикселе представлено 8-битным целым числом unsigned. Например, красный цвет имеет R = 255, G = 0, B = 0; фиолетовый: R = 255, G = 0, B = 255; серый: R = 127, G = 127, B = 127.
Используется еще 4-е значение, которое называется A. «A» от англ. «Alfa», которое отвечает за прозрачность. Если A = 0, то цвет полностью прозрачный. Если A = 255, то цвет непрозрачный.
В совокупности R, G, B и A составляют одно 32-битное целое число, с 8 битами для каждого компонента:

32-битное значение RGBA

31-24 бита	23-16 бит	15-8 бит	7-0 бит
RRRRRRRR	GGGGGGGG	BBBBBBBB	AAAAAAAA
red	green	blue	alpha

Следующая программа просит пользователя ввести 32-битное шестнадцатеричное значение, а затем извлекает 8-битные цветовые значения R, G, B и A:

```
#include <iostream>

int main()
{
    const unsigned int redBits = 0xFF000000;
    const unsigned int greenBits = 0x00FF0000;
    const unsigned int blueBits = 0x0000FF00;
    const unsigned int alphaBits = 0x000000FF;

    std::cout «
             "Enter a 32-bit RGBA color value in hexadecimal (e.g. FF7F3300): ";
    unsigned int pixel;
    std::cin >> std::hex >> pixel; /* std::hex позволяет вводить
шестнадцатеричные значения */

    /* Используем побитовое И для изоляции красных пикселей, а затем сдвигаем
значение вправо в диапазон 0-255 */
    unsigned char red = (pixel & redBits) >> 24;
    unsigned char green = (pixel & greenBits) >> 16;
    unsigned char blue = (pixel & blueBits) >> 8;
    unsigned char alpha = pixel & alphaBits;

    std::cout << "Your color contains:\n";
    std::cout << static_cast<int>(red) << " of 255 red\n";
    std::cout << static_cast<int>(green) << " of 255 green\n";
    std::cout << static_cast<int>(blue) << " of 255 blue\n";
    std::cout << static_cast<int>(alpha) << " of 255 alpha\n";

    return 0;
}
```

Результат выполнения программы:

Enter a 32-bit RGBA color value in hexadecimal (e.g. FF7F3300): FF7F3300
Your color contains:
255 of 255 red
127 of 255 green
51 of 255 blue
0 of 255 alpha

В программе, приведенной выше, побитовое И используется для запроса 8-битного набора, который нас интересует, затем мы его сдвигаем вправо в диапазон от 0 до 255 для хранения и вывода.

Некоторые элементарные задачи с битовыми операциями

Запрос битового состояния:

```
if (myflags & option4) ... // если установлен option4, то делаем что-нибудь
```

Включение бит:

```
myflags |= option4; // включаем option4
myflags |= (option4 | option5); // включаем option4 и option5
```

Выключение бит:

```
myflags &= ~option4; // выключаем option4
myflags &= ~(option4 | option5); // выключаем option4 и option5
```

Переключение между битовыми состояниями:

```
myflags ^= option4; // включаем или выключаем option4
myflags ^= (option4 | option5); // изменяем на противоположные option4 и option5
```

Пример решения задачи

Дано число проверить на зеркальность два указанных байта этого числа.

```
#include <iostream>
#include <cstdint>

int main()
{
	uint64_t n {1325862918881594572 8ull};
	uint64_t one {0x1ull};
	int b1{6}, b2{8};
	int cobi{8};
	for(int i = 0; i < 8; i++)
	{
		if(
				((n & one<<8*(b2-1)+i)>>8*(b2-1)+i) ^
				((n & one<<8*b1-1-i)>>8*b1-1-i))
		{
			cobi--;
		}
	}
	if(8 == cobi) std::cout << "ДА" << std::endl;
	else std::cout << "НЕТ" << std::endl;
}
```

Практическая часть

Ознакомится с теоретическим текстовым материалом изложенным в пояснении к лабораторной работе.

Исследовать диапазоны применимости алгоритмов для сравнения вещественных чисел на равенство

1. В первой из разрабатываемых программ следует опытным путём исследовать применимость алгоритмов для сравнения вещественных чисел на равенство. То есть исследовать применимость функций **`approximatelyEqual`** и **`approximatelyEqualAbsRel`** для проверки двух вещественных чисел на равенство.
2. Все исходные данные необходимые для работы программы следует передавать через параметры командной строки.
3. На вход программы через параметры командной строки следует подать: наименование функции, пару вещественных чисел, а также значение `epsilon` для функции **`approximatelyEqual`** либо значения `absEpsilon` и `relEpsilon` для функции **`approximatelyEqualAbsRel`**.
4. Программа должна самостоятельно определять какая именно функция из двух исследуемых должна быть использована (либо **`approximatelyEqual`** либо **`approximatelyEqualAbsRel`**).
5. Предусмотреть проверку *корректности* пользовательского ввода: через первый параметр ожидается получение строки, а через последующие — получение вещественных чисел.
6. Обеспечить в разрабатываемой программе защиту от некорректного пользовательского ввода. При этом следует продумать возможные случаи такого некорректного ввода. Перечислить в отчёте примеры возможных вариантов некорректного пользовательского ввода. Продемонстрировать то, каким образом разрабатываемая программа обрабатывает некорректный пользовательский ввод. Достаточно привести по одному примеру на один случай некорректного пользовательского ввода.[20]
7. Запускаемая программа при разных исходных данных должна демонстрировать:[21]
   1) как успешность, так и неуспешность работы указанной функции сравнения для заданных близких к нулю вещественных числах.
   2) как успешность, так и неуспешность работы указанной функции сравнения для заданных вещественных чисел близких к единице.
   3) как успешность, так и неуспешность работы указанной функции сравнения для заданных вещественных чисел много больших единицы.
8. Результат сравнения указанных чисел следует выводить в терминал (на консоль).

20 Включить в отчёт по лабораторной работе.

21 Варианты запуска программы и получаемые при этом результаты следует включить в отчёт по лабораторной работе.

9. Для нечётных номеров в списке группы использовать в исследовании версии функций `approximatelyEqual` и `approximatelyEqualAbsRel`, работающих с данными типа `float`, а для чётных — `double`.

10. Пример ввода и вывода:[22]

```
> C:\>MyProgram approximatelyEqualAbsRel 1.2e-50 1.1e-50 1e-6 1e-3
> Числа 1.2e-50 и 1.1e-50 равны.

> C:\>MyProgram approximatelyEqualAbsRel 1.2e-50 1.1e-50 1e-66 1e-63
> Числа 1.2e-50 и 1.1e-50 неравны.
```

Использование битовых операций

1. Получить задание у преподавателя (два номера задач из списка «`Задачи: использование битовых операций`»).
2. Во-второй из разрабатываемых программ следует продемонстрировать обработку и/или преобразование заданных пользователем целых беззнаковых чисел. При этом все обработки и/или преобразования чисел допускается осуществлять только с помощью битовых операций (`| , &, ~, ^, <<, >>`). В ряде заданий допускается использовать массивы, но только в качестве хранилища исходных чисел, а не для их непосредственной обработки и преобразования. Например, нельзя занести цифры числа (в произвольной системе счисления) в отдельные элементы массива (контейнера) и затем преобразовать для обеспечения решения поставленной задачи.
3. Все исходные данные необходимые для работы программы следует передавать через параметры командной строки.[23]
4. Результат работы программы следует выводить в терминал (на консоль).[24]
5. Обеспечить в разрабатываемой программе защиту от некорректного пользовательского ввода. При этом следует продумать возможные случаи такого некорректного ввода. Перечислить в отчёте примеры возможных вариантов некорректного пользовательского ввода. Продемонстрировать то, каким образом разрабатываемая программа обрабатывает некорректный пользовательский ввод. Достаточно привести по одному примеру на один случай некорректного пользовательского ввода.[25]
6. Решение каждой задачи оформить как отдельную функцию.
7. Для хранения обрабатываемых данных использовать целые беззнаковые числа.
8. Обеспечить возможность работы реализованных функций с целыми числами различных размерностей (рекомендуется использовать 8-ми байтовые беззнаковые числа типа `uint64_t` либо `unsigned long long`).
9. Оформить программу как многофайловую (`все следующие в данном пункте требования – отмеченные * – должны быть выполнены`).

---

22 Для вывода чисел с повышенной точностью используйте следующую предустановку:
std::cout << std::fixed << std::setprecision(n); , где n — требуемое количество значащих разрядов.

23 Включить в отчёт по лабораторной работе.

24 Включить в отчёт по лабораторной работе.

25 Включить в отчёт по лабораторной работе.

* Используемые константы расположить в отдельном заголовочном файле `*.h`. Например, можно разместить в нём используемые битовые маски.
* Дополнительно поместить используемые константы в отдельное пространство имён.
* Прототипы пользовательских функций расположить в отдельном заголовочном файле(файлах) `*.h`.
* Пользовательские функций расположить в отдельном файле(файлах) `*.cpp`.
* В заголовочных файлах использовать механизм `Header guards` или `#pragma once`.
* Таким образом, в главном файле проекта должна располагаться только функция `main`, в которой подключается ряд пользовательских заголовочных файлов и осуществляется вызов пользовательских функций.

10. Пример ввода и вывода для задачи №33. «Сравнение указанных последовательностей битов в двух разных заданных числах»:
> `C:\>MyProgram 33`[26] `14411518807585587209`[27] `63 60`[28] `1236950581257`[29] `41 38`[30]
> `Указанные битовые последовательности СОВПАДАЮТ.`

11. Оформить по выполненной лабораторной работе в рабочей тетради письменный отчёт.[31]

---

26 Номер решаемой задачи.
27 Первое из двух чисел.
28 Число 8-и байтовое, нумерация битов справа налево от 1-го до 64-го. Заданная битовая последовательность с 60-го бита числа 14411518807585587209 по 63-й бит этого числа.
29 Второе из двух чисел.
30 Число 8-и байтовое, нумерация битов справа налево от 1-го до 64-го. Заданная битовая последовательность с 38-го бита числа 1236950581257 по 41-й бит этого числа.
31 **Содержание отчёта:**
1. Задания на лабораторную работу.
2. Листинги программ.
3. Примеры выполнения программы (исходные данные + результаты).
4. Перечисление примеров возможных вариантов некорректного пользовательского ввода (начиная с лабораторной работы №3).

Примечание: Рабочая тетрадь с отчётами подписывается с указанием ФИО и номера учебной группы.

Задачи: использование битовых операций

1. Найти первые N целых чисел, у которых младший байт является зеркальным отражением следующего байта.
2. Найти первые N целых чисел, у которых старший байт является зеркальным отражением предыдущего байта.
3. Реализовать обмен местами старшего и младшего байт числа.
4. Реализовать обмен местами двух указанных байт числа.
5. Реализовать обмен местами двух указанных бит числа.
6. Реализовать циклический сдвиг числа вправо на указанное число байт.
7. Реализовать циклический сдвиг числа влево на указанное число байт.
8. Реализовать циклический сдвиг числа вправо на указанное число бит.
9. Реализовать циклический сдвиг числа влево на указанное число бит.
10. Реализовать побитный сдвиг числа влево на указанное число, но только для чётных битов.
11. Реализовать побитный сдвиг числа право на указанное число, но только для нечётных битов.
12. Реализовать циклический побитный сдвиг числа влево на указанное число бит, но только для чётных битов.
13. Реализовать циклический побитный сдвиг числа вправо на указанное число бит, но только для нечётных битов.
14. Определить позицию самой старшей единицы в битовом представлении данного целого числа.
15. Написать функцию, записывающую 0 или 1 в указанный бит данного целого числа и оставляющую остальные биты без изменения.
16. Написать функцию, записывающую данный байт в данное целое число с указанной позиции и оставляющую остальные биты без изменения, если часть битов записываемого байта не помещается, то они отбрасываются.
17. Написать функцию, записывающую данный байт в данное целое число с указанной позиции и оставляющую остальные биты без изменения, если часть битов записываемого байта не помещается, то они помещаются в начало данного числа.
18. Инвертирование в указанном числе n бит начиная с p–й позиции.
19. Определение количества единиц в битовой записи указанного числа.
20. Определение количества нулей в битовой записи указанного числа.
21. Сравнение числа единиц в двух числах, представленных в двоичном коде (в каком больше).

22. Сравнение числа нулей в двух числах, представленных в двоичном коде (в каком больше).
23. Определение наличия в числе указанной битовой последовательности.
24. Определение в указанном числе позиции первого вхождения заданной битовой последовательности.
25. Определение в указанном числе позиции последнего вхождения заданной битовой последовательности.
26. Суммарное количество единичных битов в числах заданного целочисленного массива.
27. Суммарное количество нулевых битов в числах заданного целочисленного массива.
28. Определить представляет ли собой двоичное представление числа палиндром (слева направо читается также как справа налево).
29. Определить является ли двоичное представление числа зеркальным относительно центра.
30. Сравнение указанных битов в заданном числе.
31. Сравнение указанных последовательностей битов в заданном числе.
32. Сравнение указанных битов в двух разных заданных числах.
33. Сравнение указанных последовательностей битов в двух разных заданных числах.