GeekBrains

Урок 6

Рекурсия
Вещественные типы данных

Программирование на языке Си (базовый уровень)

Оглавление

Введение

На предыдущей лекции вы узнали:

- Что такое функции
- Как создать свою функцию и передавать аргументы в нее
- Что такое ASCII - таблица символов
- Что такое буферизированный и не буферизированный ввод
- Про функции ввода-вывода getchar(), putchar() и getch()
- Модификатор static переменных
- Вычисление корней квадратного уравнения: добавляем функции и рефакторим код
- Пример машины состояний: “Конечный автомат для варки кофе”

На этой лекции вы найдете ответы на такие вопросы как / узнаете:

- Что такое рекурсия, её достоинства и недостатки
- Какие бывают рекурсии
- Как происходит замена цикла на рекурсию и наоборот
- Практические примеры рекурсий
- Как размещаются в памяти вещественные числа
- Разберем примеры

Термины, используемые в лекции

Рекурсия — это использование определения какого-то понятие через само это понятие.

Рекурсивная функция — вызов функции из нее же самой.

Итерация — организация обработки данных, при которой действия повторяются многократно, не приводя при этом к вызовам самих себя (в отличие от рекурсии).

Сложная рекурсия — это тип рекурсии, при котором функция A вызывает функцию B, а та в свою очередь вызывает A.

Рекурсия в функциях

Рекурсия - это использование определения какого-то понятие через само это понятие.

Например, вычисления факториала можно определить так:

n! = n * (n - 1), при n > 0 иначе 1

Рекурсия с точки зрения математики — механизм, в котором для решения задачи из функции вызывается та же самая функция.

Итерационная версия функции factorial(), предполагает использование цикла и нам более привычна. Начиная с 1 и заканчивая указанным значением n. Тут происходит последовательное умножение ранее полученного произведения на каждое число.

Вот так факториал будет описан без рекурсии:

Действие итерационной версии `factorial()` вычисления факториала очевидно. Она использует цикл, начиная с 1 и заканчивая указанным числом, последовательно перемножая каждое число на ранее полученное произведение.

Факториал можно описать также и без рекурсии

```
unsigned int factorial(unsigned int n) {
    unsigned long long fact = 1;
    for(int i = 2;i < =n;i++)
        fact *= i;
    return fact;
```

```
}
```

Кстати, есть алгоритмы, которые работают намного быстрее данной реализации: алгоритм вычисления деревом, алгоритм вычисления факторизацией.

Рассмотрим пример рекурсивной функции вычисления факториала

Рекурсивная функция - вызов функции из нее же самой.

Рекурсивная функция всегда должна состоять из:

1. Условие остановки
2. Шаг рекурсии

🔥 **Внимание!** Рекурсивная функция всегда должна иметь условие остановки

```
unsigned int factorial(unsigned int n) {
    if(n <= 1) // Условие остановки
        return 1;
    return n * factorial(n - 1); // Шаг
}
```

При вычислении факториала рекурсивной функцией алгоритм будет сложнее. Если вызвать функцию factorial() с параметром 1, то результат будет 1. Если параметр будет больше 1, то функция возвращает factorial(n - 1) * n. Теперь функция factorial() будет вызвана снова с параметром (n - 1). И так до тех пор пока n не станет единицей. Все промежуточные значения n будут сохранены в стеке.

Действие рекурсивной функции `factorial()` немного более сложно. Когда `factorial()` вызывается с аргументом 1, функция возвращает 1. В противном случае она возвращает произведение `factorial(n - 1) * n`. Для вычисления этого значения `factorial()` вызывается с n - 1. Это происходит, пока n не станет равно 1.

Если мы будем вычислять факториал числа 2, то ход вычислений будет такой:

- ➔ Первый вызов функции **`factorial()`** с параметром 2, после проверки 2<=1, приводит к повторному вызову функции **`factorial()`** с параметром 1.
- ➔ Второй вызов функции **`factorial()`**, после проверки 1<=1, где условие выполнится возвращает 1, после чего результат умножается на 2 (начальное значение n).
- ➔ Получаем ответ 2.

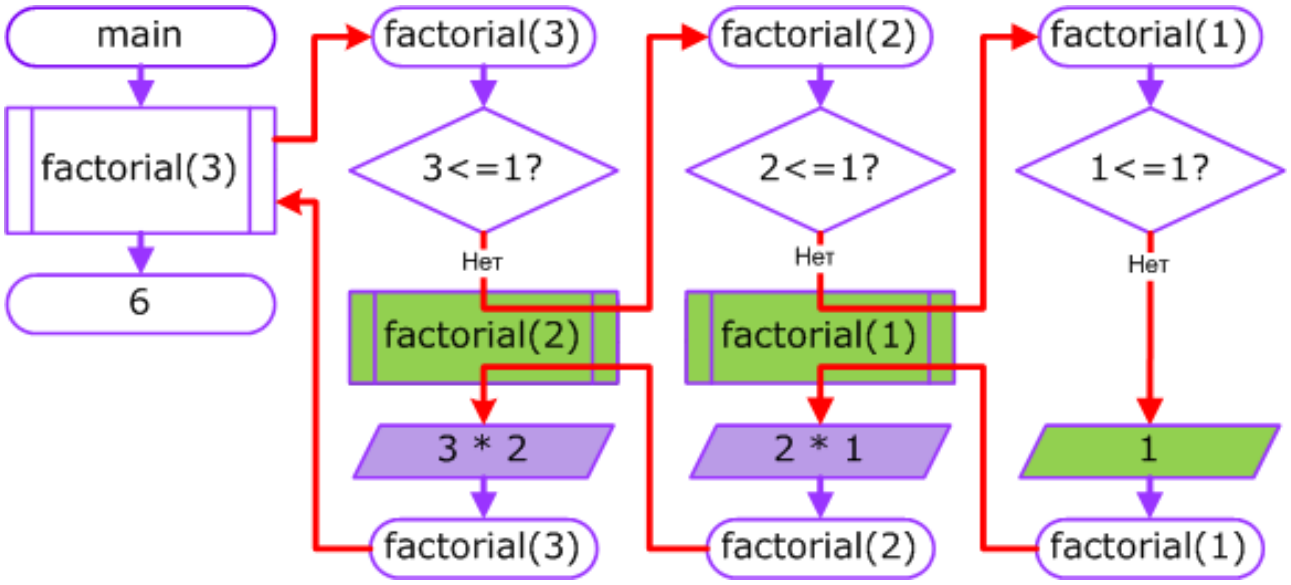
При вычислении факториала числа 2, первый вызов `factorial()` приводит ко второму вызову с аргументом 1. Данный вызов возвращает 1, после чего результат умножается на 2 (исходное значение n). Ответ, таким образом, будет 2.

Можно попробовать вставить printf() в `factorial()` для демонстрации уровней и промежуточных ответов каждого вызова.

```
unsigned int factorial(unsigned int n)
{
    printf("%d\n",n);
    if(n <= 1) // Условие остановки
        return 1;
    int _f = n * factorial(n - 1);
    printf("%d*factorial(%d)=%d\n",n,n - 1,_f);
    return _f; // Шаг
}
```

Промоделируем вызов `factorial(3)`:

`int main() {`
`  factorial(3);`
`  return 0;`
`}`
	`factorial(3) {`
	`  if(3<=1)`
	`  return 1;`
	`return`
	`3*factorial(2);`
	`}`
		`factorial(2) {`
		`  if(2<=1)`
		`  return 1;`
		`return`
		`2*factorial(1);`
		`}`
			`factorial(3) {`
			`  if(1<=1)`
			`  return 1;`
			`return`
			`n*factorial(n-1);`
			`}`



Стек вызовов

Параметры и локальные переменные функции хранятся в стеке. При каждом новом вызове функции в стеке выделяется память для новых переменных. Таким образом,

пространство стека может заполниться. Если это произойдет, то наступит ошибка переполнения стека — “stack overflow” . Это особенно актуально для IoT.

При использовании рекурсивной функции добавляются накладные расходы, поскольку функция вызывается несколько раз. Таким образом, рекурсивная версия функции зачастую будет выполняться немного медленнее, чем ее итеративные эквиваленты. Это не так важно, учитывая скорость современной вычислительной техники. Но при разработке микроконтроллеров такая разница может быть критичной.

Рекурсивные версии большинства подпрограмм могут выполняться слегка медленнее, чем их итеративные эквиваленты, поскольку добавляются накладные расходы в виде вызова функций. Но при темпах роста скорости вычислительной техники это не имеет значения, хотя для микроконтроллеров это может быть критично.

Поскольку местом для хранения параметров и локальных переменных функции является стек и каждый новый вызов создает новую копию переменных и пространство стека может исчерпаться. Если это произойдет, то возникнет ошибка — переполнение стека. Так называемый “stack overflow”

🔥 **Внимание!** Много рекурсивных вызовов в функции может привести к переполнению стека.

🔥

Рассмотрим пример рекурсивной печати четырех чисел в прямом порядке :

```
#include <stdio.h>
void Rec(int n) {
    if(n > 0)
        Rec(n - 1);
    printf("%5d",n);
}
int main(void)
{
    Rec(4);
    return 0;
}
```

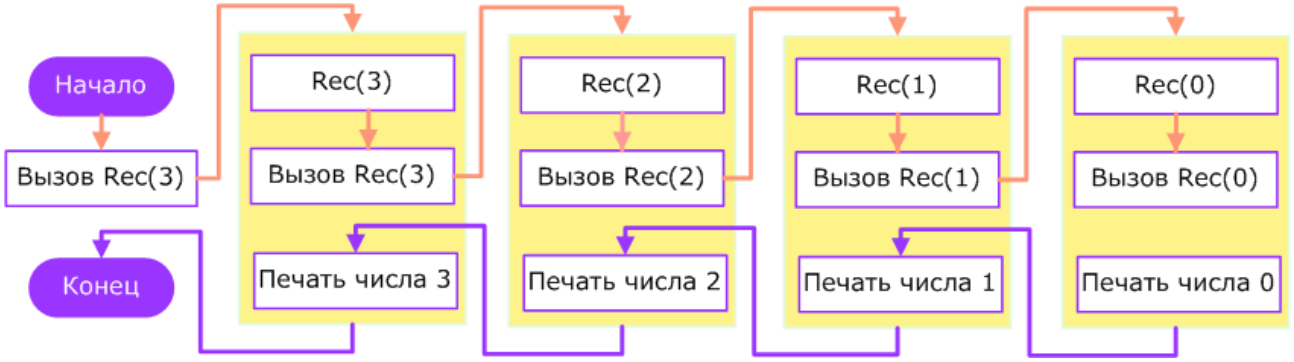
При рекурсивной печати четырех чисел от 0 до трех будут выполнено следующее:

Первый вызов функции Rec(3) с параметром 3, после проверки 3>0, приводит к повторному вызову функции Rec(2) с параметром 2.

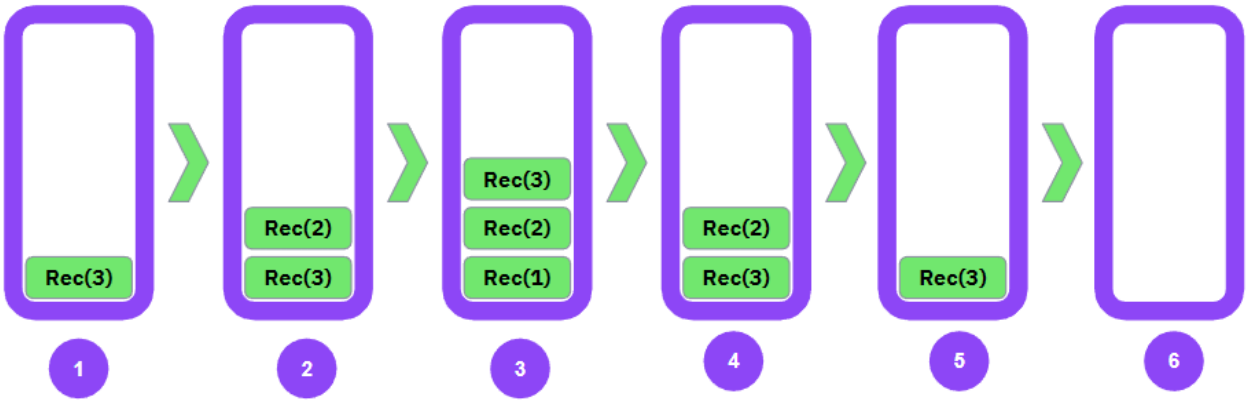
Второй вызов функции Rec(2), после проверки 2>0, приводит к повторному вызову функции Rec(1) с параметром 1

Третий вызов функции Rec(2), после проверки 3>0, приводит к повторному вызову функции Rec(0) с параметром 0

Четвертый функции Rec(0), после проверки 0>0, печатает число 0, после чего печатает число 1,2 и 3 соответственно.



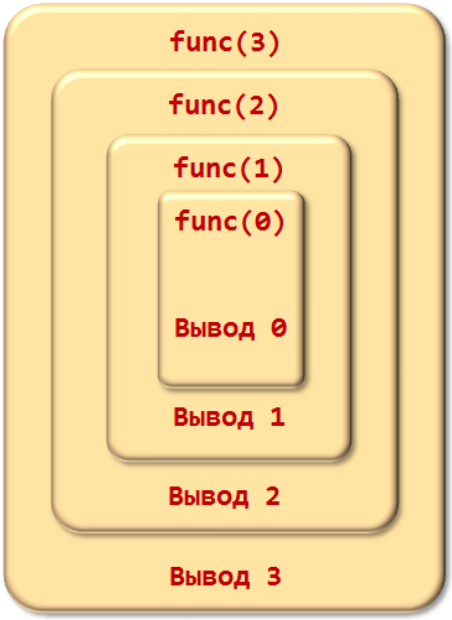
Когда функция вызывает сама себя, в стеке выделяется место для новых локальных переменных и параметров.

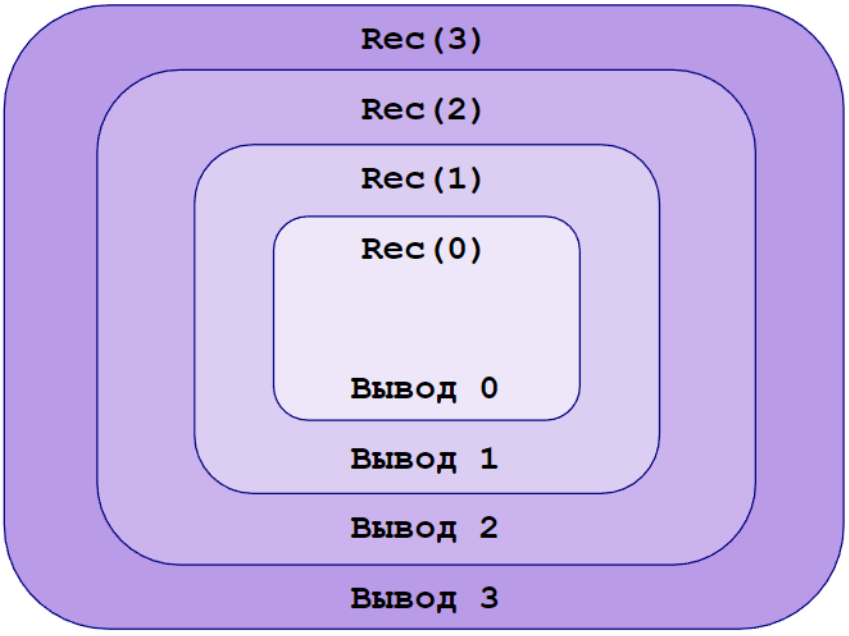


Каждая вновь вызванная функция работает со своими параметрами и локальными переменными. Под такие переменные в стеке при каждом новом вызове функции выделяется память. Когда каждая рекурсивная функция завершает работу старые переменные удаляются. Выполнение продолжается с того места, где было обращение внутри этой функции. Рекурсивные функции вкладываются одна в другую как элементы подзорной трубы.

Код функции работает с данными переменными. Рекурсивный вызов не создает новую копию функции. Новыми являются только аргументы. Поскольку каждая

рекурсивно вызванная функция завершает работу, то старые локальные переменные и параметры удаляются из стека и выполнение продолжается с точки, в которой было обращение внутри этой же функции. Рекурсивные функции вкладываются одна в другую как элементы подзорной трубы.





Префиксная и постфиксная форма записи

Если процедура вызывает сама себя, то, по сути, это приводит к повторному выполнению содержащихся в ней инструкций, что аналогично работе цикла. При этом различают префиксную и постфиксную формы записи.

Рассмотрим пример рекурсивной печати четырех чисел: при префиксной форме будет в прямом порядке, при постфиксной в обратном порядке:

//Префиксная форма //Сначала - рекурсивный вызов, потом - действия	//Постфиксная форма //Сначала - действия, потом - рекурсивный вызов

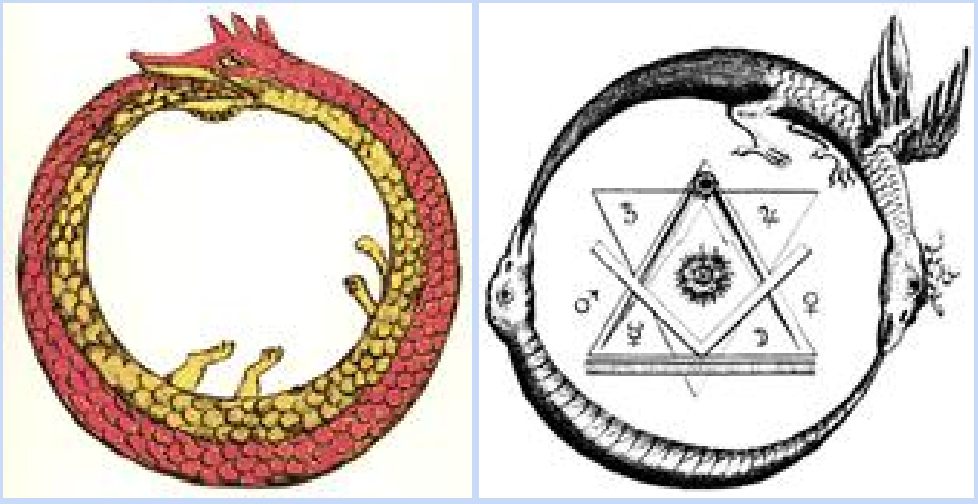
#include <stdio.h>
void Rec(int n) {
    if(n > 0)
        Rec(n - 1);
    printf("%5d",n);
}
int main(void)
{
    Rec(4);
    return 0;
}
	#include <stdio.h>
	void Rec(int n) {
	    printf("%5d",n);
	    if(n > 0)
	        Rec(n - 1);
	}
	int main(void)
	{
	    Rec(4);
	    return 0;
	}

Вывод на консоль 0 1 2 3	Вывод на консоль 3 2 1 0

Сложная рекурсия

Давайте рассмотрим теперь более сложный вариант, когда функция A(), вызывает функцию B(). В свою очередь B() вызывает A(). Это называется сложной рекурсией. Получается, что процедура, которая описана первой вызывает вторую, которая еще не описана. Поэтому для функции B() используют объявление через прототип функции (указывают заголовок без тела функции).

Сложная рекурсия — это тип рекурсии, при котором функция A вызывает функцию B, а та в свою очередь вызывает A.



Возможна чуть более сложная схема: функция *A* вызывает функцию *B*, а та в свою очередь вызывает *A*. Это называется **сложной рекурсией**. При этом оказывается, что описываемая первой процедура должна вызывать еще не описанную. Чтобы это было возможно, требуется использовать прототип, т.е. описание функции *B* до ее использования.

Пример: вычислить значение выражения

$$\frac{x^n}{n!}$$

```
#include <stdio.h>

int my_pow(int, int); // описание прототипа my_pow
double calc(int x, int n)
{
  return (double)my_pow(x, n) / n; // вызов функции pow
}
int my_pow(int x, int n)
{
  if (n == 1) return x;
  return x*calc(x, n - 1); // вызов функции calc
}
int main()
{
  int n, x;
  printf("n = ");
  scanf("%d",&n);
  printf("x = ");
  scanf("%d",&x);
  double a = calc(x, n); // вызов рекурсивной функции
```

```
  printf("%lf",a);
  return 0;
}
```

Результат выполнения программы:

n = 2
x = 3
4.500000

Рекурсия и итерирование

Итерация — организация обработки данных, при которой действия повторяются многократно, не приводя при этом к вызовам самих себя (в отличие от рекурсии).

Рассмотрим вычисление суммы цифр натурального числа в виде итерационной и рекурсивной процедуры.

```
//Нерекурсивный способ:
int sumIter(int num)
{
int sum = 0;
  while(num > 0) {
    sum = sum + num % 10;
    num = num / 10;
  }
  return sum;
}
```

```
//Рекурсивный способ:
int sumRec(int num)
{
  if (num > 0)
    return num%10 + sumRec(num/10);
  else
    return 0;
}
```

Что лучше рекурсия или итерирование?

Выбирая какой способ применить для решения задачи стоит обратить внимание на особенности задачи которую надо решить. А также на сильные и слабые стороны самих подходов.

Когда следующее вычисления полностью зависит от результата предыдущего (например, факториал) цикл будет более эффективен, поскольку это сильно сэкономит ресурсы на запоминание и хранение всех промежуточных результатов.

Рекурсия же будет эффективна, когда имеем дело с вложенными каталогами, деревьями. Если у каждой вложенной единицы есть ряд отдельных переменных (например, количество файлов в данном каталоге). Обход одного каталога совсем не зависит от обхода соседнего и они могут работать параллельно, независимо друг от друга.

Также рекурсия будет более эффективна, если рекурсивная функция кэшируемая, например, она запоминает результат и при следующем запросе просто возвращается кэшированный вариант.

Все зависит от задачи. В цикле лучше решать задачи, где результат следующего полностью зависит от результата предыдущего (факториал).

При обходе вложенных каталогов, например, когда этом у каждого имеется ряд своих отдельных переменных (например, количество файлов в данном каталоге), то поддерживать легче будет рекурсию. Да и рекурсия в данном случае будет удобнее, потому что обход одного каталога совсем не зависит от результатов обхода другого соседнего каталога, и они могут работать параллельно, независимо друг от друга.

Также рекурсия будет более эффективна, если рекурсивная функция кэшируемая, например, она запоминает результат и при следующем запросе просто возвращается кэшированный вариант.

При сравнении рекурсивного и итерационного подхода, следует отметить, достоинства рекурсии и итерационного подхода.

Плюсы рекурсии

- ➔ наглядность
- ➔ компактность

Особенно это заметно в задачах на обработку структур данных, заключающих в себе рекурсию, таких как списки, деревья и т.д.

Плюсы итерации

- ➔ экономия памяти
- ➔ быстродействие

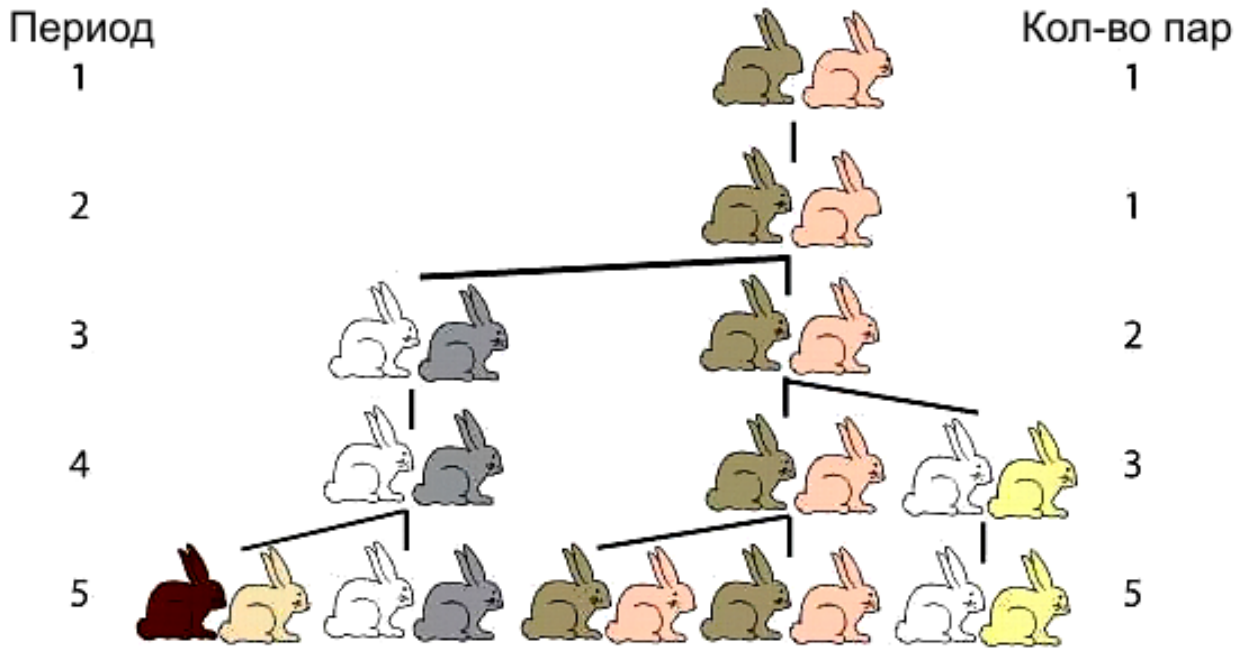
Однако за счет необходимости выделения памяти и накладных расходов процессора при вызове рекурсивной функции рекурсия требует больше памяти и она медленнее.

Подитог. Сравнивая рекурсивный и итеративный подход, следует отметить, что несомненным достоинством рекурсивной реализации является ее наглядность и компактность. Особенно хорошо это видно при обработке структур данных, предполагающих внутри себя рекурсию, таких как списки, деревья и т.д. Однако за счет накладных расходов на обеспечение рекурсивных вызовов функции рекурсивная реализация проигрывает итеративной в эффективности как по памяти, так и по быстродействию.

Проблемы рекурсии

Числа Фибоначчи в Европе популяризовал Леонардо Пизанский (по прозвищу Фибоначчи – сын Боначчи), в задаче о кроликах:

Пусть в огороженном месте имеется пара кроликов (самка и самец) в первый день января. Эта пара кроликов производит новую пару кроликов (самку и самца) в первый день февраля и затем в первый день каждого следующего месяца. Каждая новорожденная пара кроликов становится зрелой уже через месяц и затем через месяц дает жизнь новой паре кроликов. Возникает вопрос: сколько пар кроликов будет в огороженном месте через год, то есть через 12 месяцев с начала размножения.



Оказывается, число кроликов по месяцам описывается последовательностью

1, 2, 3, 5, 8, 13,...

В ней каждое число равно сумме двух предыдущих. Как мы видим, последовательность очень быстро растет

Вычислить рекурсивно число Фибоначчи N очень легко.

```c
#include <stdio.h>
int fibonachi(int n) {
    if(n<=0)
        return 0;
    if(n==1)
        return 1;
```

8

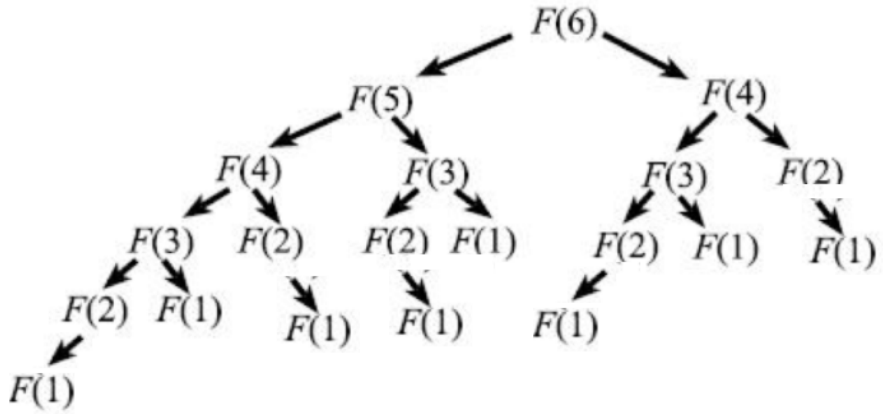
```
    return
fibonachi(n-1)+fibonachi(n-2);
}
int main(void)
{

printf("%d\n",fibonachi(6));
}
```

Вставим printf() в fibonachi() для демонстрации уровней и промежуточных ответов каждого вызова.

```
#include <stdio.h>
int fibonachi(int n)
{
    printf("n=%d\n",n);
    if(n <= 0)
    {
        //printf("F(%d)=0\n",n);
        printf("return 0\n");
        return 0;
    }
    if(n==1)
    {
        printf("F(1)=1\n");
        //printf("F(%d)=1\n",n);
        return 1;
    }
    int Fib = fibonachi(n - 1)+fibonachi(n - 2);
    printf("Fib(%d)=%d\n",n,Fib);
    return Fib;
}
int main(void)
{
    printf("%d\n",fibonachi(6));
    return 0;
}
```

Как видно из картинки, одно и тоже действие будет выполнено несколько раз. Как можно было бы исправить данную проблему?



- Сделать итерационную версию
- Кешировать результат, например в глобальном массиве.

Хвостовая рекурсия

Если структура рекурсивного алгоритма такова, что рекурсивный вызов является последней выполняемой операцией в функции, а его результат непосредственно возвращается в качестве результата функции, то такую рекурсию называют хвостовой. Компиляторы поддерживающие оптимизацию кода могут заменить ее на цикл.

Замена цикла на рекурсию

Мы уже видели, что некоторые рекурсивные алгоритмы – числа Фибоначчи, вставка и поиск в бинарных деревьях – имеют циклические эквиваленты. Что можно сказать в общем случае?

Фактически, нетрудно заменить любой цикл рекурсией. Рассмотрим произвольный цикл, данный здесь без указания инвариантов и варианта (хотя позже мы познакомимся с рекурсивными двойниками):

Необходимо реализовать с помощью рекурсии данный цикл:

```
for(i = 1; i < n; i++)
    printf("%d ", i);
```

Мы можем заменить его на

```
void recursionFor(int i, int n) {
    if(i == n)
        return;
    printf("%d ", i);
```

```
    recursionFor(i+1, n);
}
```

В функциональных языках (таких как Lisp, Scheme, Haskell, ML) рекурсивный стиль является предпочитаемым, даже если доступны циклы.

Задачи на рекурсию

1. На стандартном потоке ввода задан текст оканчивающаяся точкой, точка в текст не входит. На стандартный поток вывода вывести этот текст в обратном порядке.

```
void print_rev (void)
{
    char ch;
    scanf ("%c", &ch); //ввод очередного символа
    if (ch != '.') {
        print_rev (); //рекурсивный вызов для обработки
                      //оставшихся символов
        printf ("%c", ch); //вывод символа
    }
}
```

2. Описать рекурсивную функцию вычисления НОД.

```
int gcd ( int n, int m )
{
        if(n == m)
            return n ;
        if (n < m)
            return gcd(n,m - n );
        return gcd(n - m,m );
}
```

3. Написать рекурсивную функцию перевод числа в двоичную систему счисления.

```cpp
#include <iostream>
void dec_to_bin(int n)
{
  if (n >= 2)
    dec_to_bin(n / 2);
  std::cout <<  n % 2;
}

int main()
{
  int n;
  std::cout << "\n\nn -> ";
  std::cin >> n;
  std::cout << "\n\nBin = ";
  dec_to_bin(n);
  return 0;
}
```

4. Дано натуральное число N. Вычислите сумму его цифр.
5. Дано натуральное число N. Выведите все его цифры по одной, в обратном порядке, разделяя их пробелами или новыми строками.
6. Дано натуральное число N. Выведите все его цифры по одной, в обычном порядке, разделяя их пробелами или новыми строками.
7. Дано натуральное число n>1. Проверьте, является ли оно простым. Программа должна вывести слово
8. Дано натуральное число n>1. Выведите все простые множители этого числа в порядке неубывания.
9. Дана последовательность натуральных чисел, завершающаяся числом 0. Выведите все нечетные числа из этой последовательности, сохраняя их порядок.
10. Дана последовательность натуральных чисел, завершающаяся числом 0. Найдите максимум.

```cpp
uint32_t max_find() {
    uint32_t number, max;
    scanf("%u", &number);
    if(number == 0)
        return 0;
    max = max_find();
    if (max < number)
        max = number;
    return max;
}
```

Задача рекурсивное вычисления Sin через ряд Тейлора

Составить функцию, которая вычисляет синус как сумму ряда (с точностью 0.001) sin(x) = x - x^3/3! + x^5/5! - x^7/7! + ...(x в радианах)

При вычислении синуса и косинуса можно использовать разложение в ряд Тейлора

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \dots.$$

Общий член ряда выглядит так:

$$\sin z = \sum_{k=0}^{\infty} (-1)^k \frac{z^{2k+1}}{(2k+1)!},$$

Допустим, нужно вычислить значение sin(x) для указанного значения x, заданного в радианах, с заданной точностью. Точность вычисления считается выполненной, если последнее слагаемое в удовлетворяет условию:

$$\left| \frac{x^{2 \cdot n - 1}}{n!} \right| < \varepsilon$$

На каждом шаге на требуется вычислять факториал и выполнять возведение в степень — делать это напрямую очень неэффективно, гораздо лучше использовать значения, вычисленные на предыдущем шаге. В данном случае:

$$S_0 = x,$$
$$S_{k+1} = S_k \cdot \frac{(-1)^{k-1} \cdot x^{2 \cdot k - 1}}{(2 \cdot k - 1)!}$$

Алгоритм:

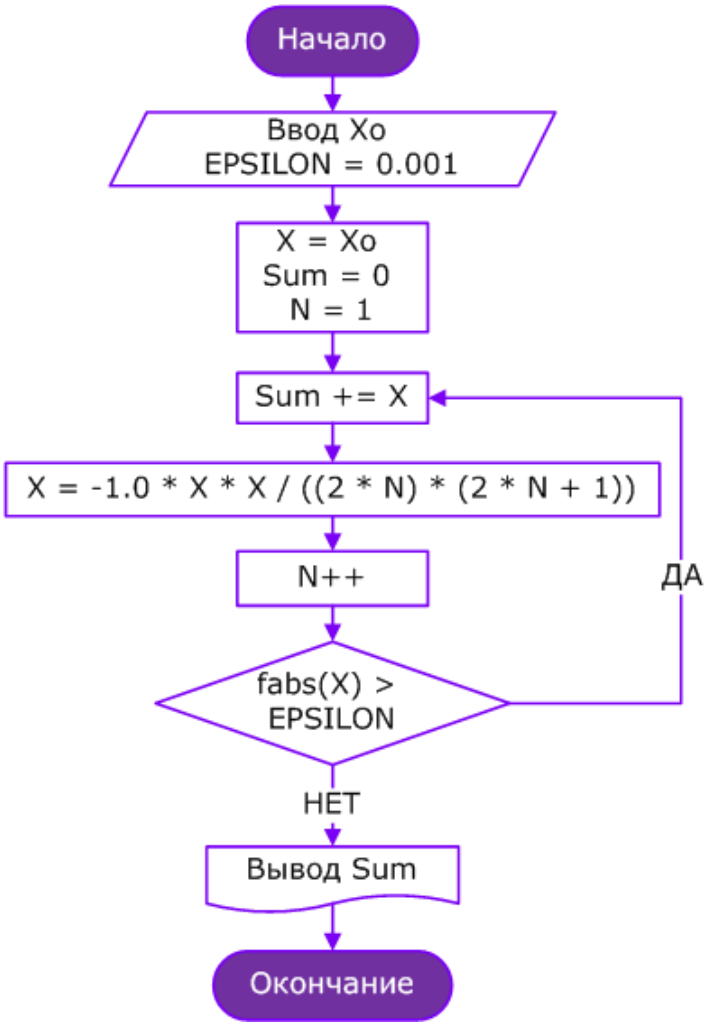
1. Ввести значение x, заданного в радианах, с точностью ε = 0.001.
2. Заводим переменную S = 0 для хранения результата подсчета ряда.
3. Заводим счетчик итераций n = 0.
4. Пока не достигнута точность (проверяем по формуле):
    a. Подсчитываем очередное значение x:
        I. Умножаем (-1) на x^2 .
        II. `T = -x^2 / 2*n.`

III. `S = T / (2*n+1)`

b. Суммируем подсчитанное значение с уже имеющимся результатом S = S + x.

c. n = n+1

5. Вывести результат вычисления sin(x), сохраненный в переменную S.



Решение задачи итерационным способом

```c
#include <stdio.h>
#include <math.h>
const float PI = 3.1415926535;
int main()
{
    double x;
    scanf("%lf",&x);
    x *= PI/180.0;
 //   printf("sinx = %lf \tlibSin =
%f",sin(x),sinx(x));
    printf("%.3f",sinx(x));
    return 0;
}
```

Функция вычисления sin через ряд Тейлора

```c
double sinx(double x)
{
    double Xn = x;
    double sum = 0.0;
    int i = 1;
    do
    {
        sum += Xn;
        Xn *= -1.0 * x * x / ((2 * i) * (2 * i + 1));
        i++;
    }
    while (fabs(Xn) > 0.001);
    return sum;
}
```

Рефакторинг кода (было) или анти паттерн программирования. Не пишите так код!

```
#include "stdafx.h"
#include <conio.h>
#include <math.h>
#include <locale.h>
#include <windows.h>
#include <string.h>

extern float END = 0.000001;
double teylor(double summ,
double z, int n);
float x;
float VAL, LASTVAL;
long float z = 1;
long float summ = 1;
float d = 1;
int n = 0;
int main()
{
setlocale( LC_ALL,"Russian" );
printf("Рекурсия. Ряд Тейлора.
Косинус.\n");
printf("Введите икс:");
scanf("%f", &x);
teylor(0,z,0);
printf("Cos = %lf\n", d);
getch();
}
```

```
double teylor(double summ, double
z, int n)
{

if (x == 0) return 1;
else if (fabs(z)>END)
{
n += 1;
z = (-1)*z*x*x/(2*n*(2*n-1));
d = d + z;
return summ=teylor(summ+d, z, n);
}

}
```

Рефакторинг кода (стало)

```
#include <math.h>
#include <locale.h>
#include <stdio.h>
float const END = 0.000001;
double teylor(double Xn, int n,
double x,double sum)
{
 if (fabs(Xn)>END)
 {
  n+=1;
  Xn*=(-1)*x*x/(2*n*(2*n-1));
  sum+=Xn;
  return teylor(Xn, n, x, sum);
 }
 return sum;
}
```

```
int main()
{
double x;
 setlocale( LC_ALL,"Russian" );
 printf("Рекурсия. Ряд Тейлора.
Косинус.\n");
 printf("Введите икс в радианах:");
 scanf("%lf", &x);
 printf("Cos = %lf\n",
teylor(1.0,0,x,1.0));
 printf("Cos = %lf\n", cos(x));
 return 0;
}
```

Представление чисел в памяти

Вспомним материал Занятия 2. Все числа в памяти компьютера хранятся в двоичном виде. Мы разделили все числа на две группы, которые в свою очередь можно разделить на подгруппы.

1. Целые числа.
   1.1. Целые числа без знака - хранятся в двоичном виде в прямом коде
   1.2. Целые числа со знаком - хранятся в двоичном виде в дополнительном коде
   1.3. Символы, коды символов - хранятся также как и беззнаковые целые числа, в двоичном коде в явном виде

2. Вещественные числа хранятся в памяти в следующем виде:
   S - знак мантиссы (0 - положительное число, 1 - отрицательное число)
   E - порядок (выражает степень основания числа, на которое умножается мантисса)
   M - мантисса (выражает значение числа без учёта порядка)

Представление вещественных чисел (IEEE 754)

Вещественные числа хранятся в нормализованном виде и денормализованном виде (маленькие по модулю). Рассмотрим пример нормализации десятичного числа 123.45. Такое число представляется в виде:
$123.45 = 1.2345 * 10^2$

Аналогично для двоичного числа - 1111.101 (15.625_{10}):
$1111.101 = \mathbf{1}.111101 * 2^3$

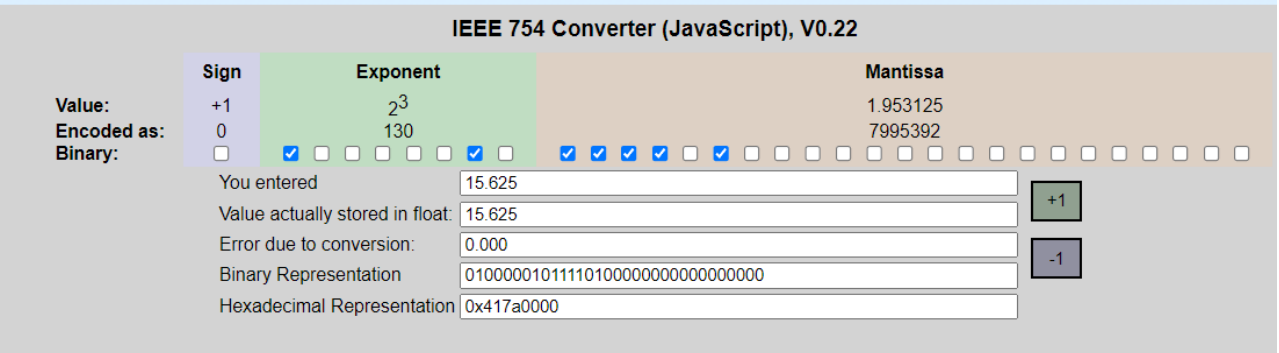
Целая часть числа мантиссы двоичного числа в нормализованном виде равен 1, поэтому при записи мантиссы числа в памяти старший разряд можно не записывать, что и используется в стандарте IEEE 754. Число 1111.101 сначала нормализуется: **1**.111101 * 2^3, **неявная единица** отбрасывается и в мантиссу сохраняется только дробная часть дополненная незначащими нулями: 111101. Такой способ позволяет сберечь один бит памяти. Порядок для нашего числа равен 3 (степень двойки). Число 3 переводится в двоичный вид и к нему прибавляется **bias** (сдвиг). Bias рассчитывается по следующей формуле:

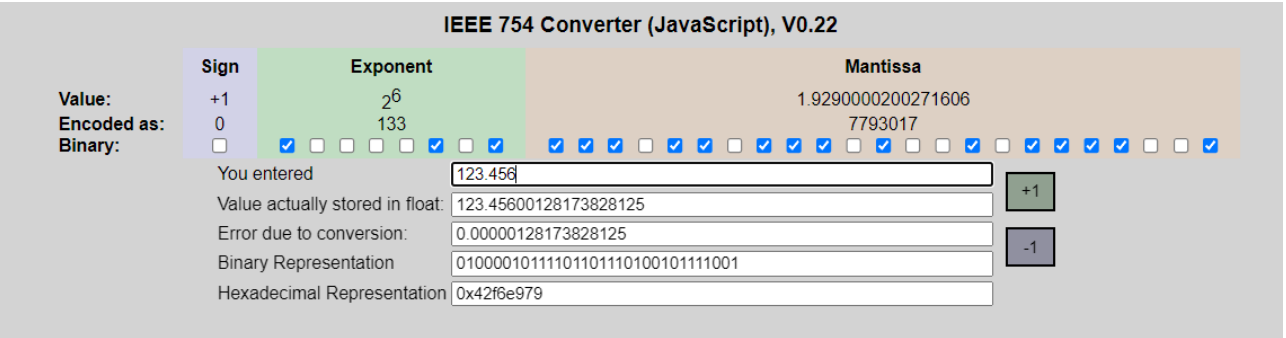
$bias = 2^{K-1} - 1$
K - количество бит выделенное под хранение порядка.

Размер выделенный под хранение вещественных чисел зависит от архитектуры ЭВМ. В языке Си определены типы

Тип	Размер	Мантисса	Порядок	Сдвиг
float	32	23	8	127
double	64	52	11	1023
long double	80	64	15	16383

В Интернете есть удобные онлайн-конвертеры IEEE 754 , например
https://www.h-schmidt.net/FloatConverter/IEEE754.html





Рассмотрим на примере, как будет хранится число 15.625 в типе float.

```
int main()
{
    /*
     тип float 4 байта
     S - знак числа 1 бит
     P - порядок 8 бит
     M - мантисса 23 бита
    */
    float f = 15.625;
```

```
    print_float_bin(f);
    return 0;
}
```

Для наглядности, используем функцию, которая отобразит наше число в двоичном виде.

```
#include <stdio.h>
//Перевод вещественного числа в двоичный вид
void print_float_bin(float num)
{
    for(int i=31;i>=0;i--)
    {
        if( i==30 || i==22)
            putchar(' ');
        if((int)num&(1<<i))//Приводим к целому типу перед сдвигом
            putchar('1');
        else
            putchar('0');
    }
    putchar('\n');
}
```

На экран будет выведено

```
0 10000010 11110100000000000000000
S     E              M
```

Если число слишком мало (по модулю), оно представляется в денормализованном виде. Поле E заполняется нулями, порядок принимается равным 1-bias , а мантисса должна в этом случае принадлежать полуинтервалу [0.0, 1.0). В поле M пишется последовательность битов, кодирующая дробную часть мантиссы (ведущий 0 отбрасывается).

Задачи

1. Как будет хранится число 0,625 тип float
2. Как будет хранится число 3,875 тип float
3. Как будет хранится число 64 тип float

Арифметические операции

- **Сложение**
  Пример: 5.5 + 3 = 101.1_2 + 11_2 = 8.5 = 1000.1_2
  1. Порядок выравнивается до большего
     5.5 = $1.011_2 * 2^2$
     3 = $1.1 * 2^1 = 0.11_2 * 2^2$
  2. Мантиссы складываются
     $1.011_2 + 0.110_2 = 10.001_2$
  3. Результат нормализуется
     $10.001_2 * 2^2 = 1.0001_2 * 2^3 = 1000.1_2 = 8.5$
- **Вычитание**
  1. Порядок выравнивается до большего
  2. Мантиссы вычитаются
  3. Результат нормализуется
- **Умножение**
  1. Мантиссы умножаются
  2. Порядки складываются
  3. Результат нормализуется
- **Деление**
  1. Мантиссы делятся
  2. Порядки вычитаются
  3. Результат нормализуется

Погрешность вещественных чисел

Многие десятичные дроби невозможно представить в виде конечной двоичной дроби, например:
0.1_{10} = $0.0(0011)_2$ - бесконечная двоичная дробь.

Для хранения большинства дробных чисел требуется бесконечное число разрядов. Рассмотрим пример. В возьмем переменную one=1, а в переменную on_error, в цикле, десять раз будем прибавлять 0.1. В итоге в переменной one_error тоже должна оказаться 1, но это не совсем так, из-за накопившейся ошибки числа отличаются в самом последнем двоичном разряде.

```
float one=1, one_error=0;
int i;
print_float_bin(0.1);
// 10 раз прибавляем 0.1
```

```
0 01111011
10011001100110011001101
```

```
for(i=0;i<10;i++,one_error+=0.1
);
    if ( one == one_error )
        printf("Yes\n");
    else
        printf("No\n");
    printf("one = %f\n", one);
    printf("one_error = %f\n",
one_error);

    print_float_bin(one);
    print_float_bin(one_error);

    return 0;
```

```
No
one = 1.000000
on_error = 1.0000000

0 01111111
00000000000000000000000
0 01111111
00000000000000000000001
```

🔥 **Внимание!** Большинство вещественных чисел хранится в памяти с ошибкой. Будьте осторожны при их сравнении

```
//Так не надо
    float a,b;
    scanf("%f%f",&a,&b);

    if (a==b)
        printf("Yes");
    else
        printf("No");
```

```
//Используйте epselon
    float a,b;
    scanf("%f%f",&a,&b);

    if ( fabs(a-b) < 0.0001 )
        printf("Yes");
    else
        printf("No");
```

Проблемы округления, применительно к радиолокации

Вечером 25 февраля 1991 года, уже под самый конец операции «Буря в пустыне», на американскую авиабазу в саудовском Дахране свалился иракский «Скад». Вот, казалось бы, причём здесь округление дробей?

Внутренний таймер ЗРК Patriot устроен как счетчик количества интервалов времени, прошедшего с момента включения системы.

Длина такого интервала — 0,1 секунды. Чтобы перевести количество этих отрезков в секунды, его, нужно разделить на 10. Разработчики предложили умножить на 0,1.

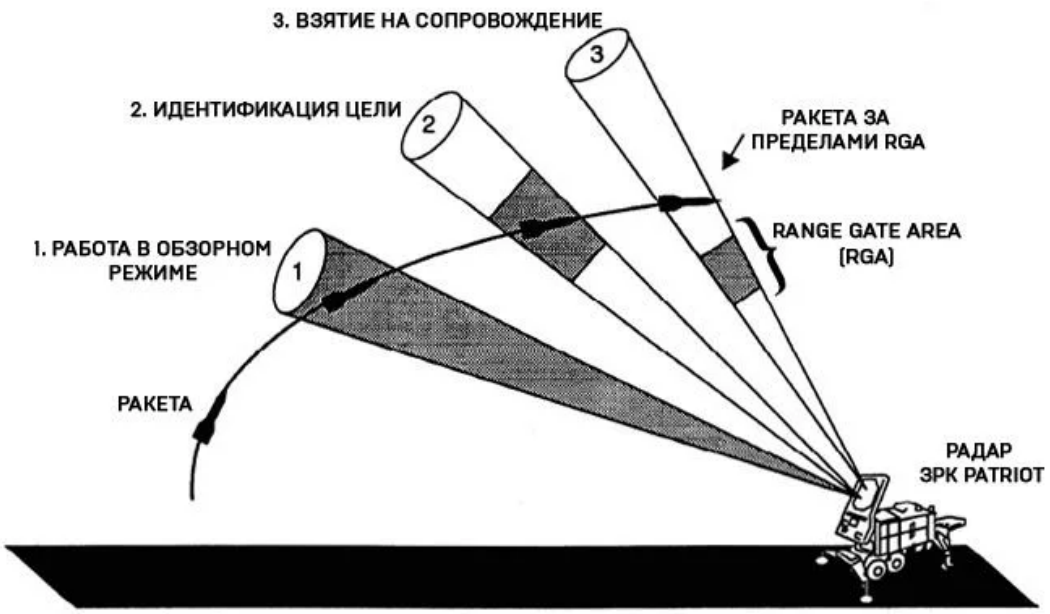
Точного представления десятичной дроби 0,1 в двоичном виде не существует — оно может быть только приблизительным.

Вместо десятичного 0,1 в систему двоичное число 0,0001100110011001100110011001100. Оно немногим меньше требуемых 0,1 — примерно на одну десятимиллионную

При расчёте параметров движения цели система оперирует близкими значениями времени с единой и очень небольшой систематической погрешностью.

При переделке батареи Patriot для Ближнего Востока, потребовалось чтобы те могли перехватывать баллистические цели, идущие со значительной скоростью — 1700 м/с и больше.

Теперь погрешности в математике ЗРК уже начали что-то решать.



Подведение итогов

Итак, на этой лекции мы начали с изучили рекурсию в функциях, как работает стек вызовов при рекурсии.

Рассмотрели префиксная и постфиксная форма записи и сложную рекурсию.

Сравнили рекурсия и итерирование и что лучше?

Также отметили проблемы рекурсии, что такое хвостовая рекурсия и замена цикла на рекурсию.

Порешали задачи на рекурсию, среди которых хочется особо отметить рекурсивное вычисления Sin через ряд Тейлора.

Затронули вопросы представления чисел в памяти и в частности представление вещественных чисел по стандарту IEEE 754, а также арифметические операции с ними.

Узнали про погрешность вещественных чисел, и каким последствиям она может привести.

Дополнительные материалы

1. Домашнее задание задачи B1 - B21
2. Оформление программного кода http://www.stolyarov.info/books/pdf/codestyle2.pdf
3. Стандарт разработки программного обеспечения MISRA на языке C http://easyelectronics.ru/files/Book/misra_c_rus.pdf
4. https://habr.com/ru/articles/337030/ Как работает рекурсия – объяснение в блок-схемах и видео
5. https://metanit.com/cpp/tutorial/3.6.php Рекурсивные функции
6. https://studfile.net/preview/1562272/page:24/ Рекурсивные функции
7. https://www.bestprog.net/ru/2019/01/07/recursion-examples-of-tasks-solving-advantages-and-disadvantages-of-recursion-ru-2/ Рекурсия. Примеры решения задач. Преимущества и недостатки рекурсии
8. http://www.c-cpp.ru/books/rekursiya Рекурсия
9. https://prog-cpp.ru/recursion/ Рекурсия
10. https://dzen.ru/a/WmWkLdyvjsv7gL3V Как один маленький баг угробил 28 американцев
11. https://www.h-schmidt.net/FloatConverter/IEEE754.html IEEE-754 Floating Point Converter
12. https://techrocks.ru/2020/02/16/recursion-and-the-call-stack-explained/ Простое объяснение рекурсии и стека вызовов
13. https://www.bestprog.net/ru/2019/01/07/recursion-examples-of-tasks-solving-advantages-and-disadvantages-of-recursion-ru-2/ Рекурсия. Примеры решения задач. Преимущества и недостатки рекурсии

Используемые источники

1. cprogramming.com - учебники по C и C++
2. free-programming-books - ресурс содержащий множество книг и статей по программированию, в том числе по C и C++ (там же можно найти ссылку на распространяемую бесплатно автором html-версию книги Eckel B. «Thinking in C++»)
3. tutorialspoint.com - ещё один ресурс с множеством руководств по изучению различных языков и технологий, в том числе содержит учебники по C
4. Юричев Д. «Заметки о языке программирования Си/Си++» - «для тех, кто хочет освежить свои знания по Си/Си++»
5. Онлайн версия «Си для встраиваемых систем»