top
КОМПЬЮТЕРНАЯ АКАДЕМИЯ

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C++

Урок № 11

Рекурсия, быстрая сортировка

Содержание

Материалы урока прикреплены к данному PDF-файлу. Для доступа к материалам, урок необходимо открыть в программе Adobe Acrobat Reader.

1. Знакомство с рекурсией

Рекурсия — это прием программирования, при котором функция вызывает саму себя либо непосредственно, либо косвенно.

Как правило, неопытный программист, узнав про рекурсию, испытывает легкое недоумение. Первая мысль — это бессмысленно!!! Такой ряд вызовов превратится в вечный цикл, похожий на змею, которая съела сама себя, или приведет к ошибке на этапе выполнения, когда программа поглотит все ресурсы памяти.

Однако рекурсия — это превосходный инструмент, который при умелом и правильном использовании поможет программисту решить множество сложных задач.

Пример на рекурсию

Исторически сложилось так, что в качестве первого примера на рекурсию почти всегда приводят пример вычисления факториала.

Что же, не будем нарушать традиций. Для начала, вспомним, что такое факториал. Обозначается факториал восклицательным знаком «!» и вычисляется следующим образом:

```
N! = 1 * 2 * 3 * ... * N
```

Другими словами, факториал представляет собой произведение натуральных чисел от 1 до N включительно. Исходя из вышеописанной формулы, можно обратить внимание на следующую закономерность:

```
N! = N * (N-1)!
```

Ура! Мы можем найти факториал через сам факториал! Вот здесь мы и попадаемся в ловушку. Наша находка, на первый взгляд, абсолютно бесполезна, ведь неизвестное понятие определяется через такое же неизвестное понятие, и получается бесконечный цикл. Выход из данной ситуации сразу же будет найден, если добавить к определению факториала следующий факт:

```
1!=1
```

Теперь мы можем себе позволить вычислить значение факториала любого числа. Попробуем, например, получить 5!, несколько раз применив формулу N! = N * (N–1)! и один раз формулу 1! = 1:

```
5! = 5 * 4! = 5 * 4 * 3! = 5 * 4 * 3 * 2! =
     5 * 4 * 3 * 2 * 1! = 5 * 4 * 3 * 2 * 1
```

Как же будет выглядеть данный алгоритм, если перенести его на язык C++? Давайте, попробуем реализовать рекурсивную функцию:

```
#include <iostream>
using namespace std;

long int Fact(long int N)
{
    // если произведена попытка вычислить
    // факториал нуля
    if (N < 1) return 0;
    /* если вычисляется факториал единицы
       именно здесь производится выход из рекурсии
    */
    else if (N == 1) return 1;
```

```
    // любое другое число вызывает функцию заново
    // с формулой N-1
    else return N * Fact(N-1);
}

int main()
{
    long number=5;
    // первый вызов рекурсивной функции
    long result=Fact(number);
    cout<<"Result "<<number<<"! is - "<<result<<"\n";
    return 0;
}
```

Как видите, всё не так уж сложно. Для более детального понимания примера рекомендуем скопировать текст программы в Visual Studio и пошагово пройтись по коду отладчиком.

2. Рекурсии или итерации?

Изучив предыдущий раздел урока — вы наверняка задались вопросом: а зачем нужна рекурсия? Ведь, реализовать вычисление факториала можно и с помощью итераций и это совсем не сложно:

```
#include <iostream>
using namespace std;
long int Fact2(long int N)
{
    long int F = 1;
    // цикл осуществляет подсчет факториала
    for (int i=2; i<=N; i++)
        F *= i;
    return F;
}

int main()
{
    long number=5;
    long result=Fact2(number);
    cout<<"Result "<<number<<"! is - "<<result<<"\n";
    return 0;
}
```

Такой алгоритм, наверное, будет более естественным для программистов. На самом деле, это не совсем так.

С точки зрения теории, любой алгоритм, который можно реализовать рекурсивно, совершенно спокойно реализуется итеративно. Мы только что в этом убедились.

Однако это не совсем так. Рекурсия производит вычисления гораздо медленнее, чем итерация. Кроме того,

рекурсия потребляет намного больше оперативной памяти в момент своей работы.

Значит ли это, что рекурсия бесполезна? Ни в коем случае!!! Существует ряд задач, для которых рекурсивное решение тонко и красиво, а итеративное — сложно, громоздко и неестественно. Ваша задача, в данном случае — научиться, не только оперировать рекурсией и итерацией, но и интуитивно выбирать, какой из подходов применять в конкретном случае. От себя можем сказать, что лучшее применение рекурсии — это решение задач, для которых свойственна следующая черта: решение задачи сводится к решению таких же задач, но меньшей размерности и, следовательно, гораздо легче разрешаемых.

Удачи Вам на данном поприще! Как говорится: «Для того, чтобы понять рекурсию, надо просто понять рекурсию».

3. Быстрая сортировка

«Быстрая сортировка» — была разработана около 40 лет назад и является наиболее широко применяемым и в принципе самым эффективным алгоритмом. Метод основан на разделении массива на части. Общая схема такова:

1. Из массива выбирается некоторый опорный элемент a[i].
2. Запускается функция разделения массива, которая перемещает все ключи, меньшие, либо равные a[i], слева от него, а все ключи, большие, либо равные a[i] — справа, теперь массив состоит из двух частей, причем элементы левой меньше элементов правой.
3. Если в подмассиве более двух элементов, рекурсивно запускаем для них ту же функцию.
4. В конце получится полностью отсортированная последовательность.

Рассмотрим алгоритм более детально.

Делим массив пополам

Входные данные: массив a[0]...a[N] и элемент p, по которому будет производиться разделение.

1. Введем два указателя: i и j. В начале алгоритма они указывают, соответственно, на левый и правый конец последовательности.
2. Будем двигать указатель i с шагом в 1 элемент по направлению к концу массива, пока не будет найден элемент a[i] >= p.

3. Затем аналогичным образом начнем двигать указатель j от конца массива к началу, пока не будет найден a[j] <= p.
4. Далее, если i <= j, меняем a[i] и a[j] местами и продолжаем двигать i, j по тем же правилам.
5. Повторяем шаг 3, пока i <= j.

Рассмотрим рисунок, где опорный элемент p = a[3].

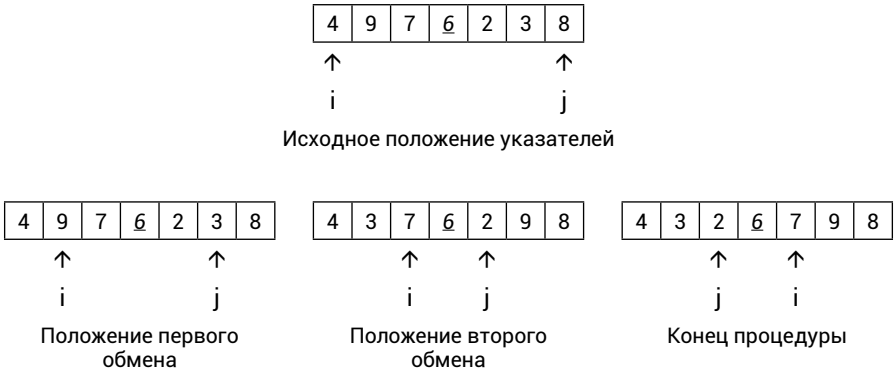

Рисунок 1

Массив разделился на две части: все элементы левой меньше либо равны p, все элементы правой — больше, либо равны p.

Пример программы

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;

template <class T>
void quickSortR(T a[], long N) {
```

```
    // На входе - массив a[],
    // a[N] - его последний элемент.
    // поставить указатели на исходные места
    long i = 0, j = N;
    T temp, p;
    p = a[ N/2 ]; // центральный элемент
                  // процедура разделения
    do {
        while ( a[i] < p ) i++;
        while ( a[j] > p ) j--;
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            i++;
            j--;
        }
    }while ( i<=j );
    /*
       рекурсивные вызовы, если есть,
       что сортировать
    */
    if ( j > 0 ) quickSortR(a, j);
    if ( N > i ) quickSortR(a+i, N-i);
}

int main(){
    srand(time(NULL));
    const long SIZE=10;
    int ar[SIZE];
    // до сортировки
    for(int i=0;i<SIZE;i++){
        ar[i]=rand()%100;
        cout<<ar[i]<<"\t";
    }
    cout<<"\n\n";
    quickSortR(ar,SIZE-1);
```

```
    // после сортировки
    for(int i=0;i<SIZE;i++){
        cout<<ar[i]<<"\t";
    }
    cout<<"\n\n";
    return 0;
}
```

Алгоритм рекурсии

1. Выбрать опорный элемент p — середину массива.
2. Разделить массив по этому элементу.
3. Если подмассив слева от p содержит более одного элемента, вызвать quickSortR для него.
4. Если подмассив справа от p содержит более одного элемента, вызвать quickSortR для него.

4. Двоичный поиск

В прошлом уроке мы рассмотрели алгоритм линейного поиска, однако это не единственная возможность организовать поиск в массиве. Если у нас есть массив, содержащий упорядоченную последовательность данных(это значит, что перед работой алгоритма двоичного поиска вам надо отсортировать исходный массив данных), то, в данном случае, очень эффективен двоичный поиск.

Теория двоичного поиска

Предположим, что переменные Lb и Ub содержат, соответственно, левую и правую границы отрезка массива, где находится нужный нам элемент. Поиск мы всегда будем начинать с анализа среднего элемента отрезка массива. Если искомое значение меньше среднего элемента, мы переходим к поиску в верхней половине отрезка, где все элементы меньше только что проверенного. Другими словами, значением Ub становится (M (средний элемент) –1) и на следующей итерации мы работаем с половиной массива. Таким образом, в результате каждой проверки мы вдвое сужаем область поиска. Так, в нашем примере, после первой итерации область поиска — всего лишь три элемента, после второй остается всего лишь один элемент. Таким образом, если длина массива равна 6, нам достаточно трех итераций, чтобы найти нужное число.

```
#include <iostream>
#include <stdlib.h>
#include <time.h>

using namespace std;
template <class T>

void bubbleSort(T a[], long size) {
    long i, j;
    T x;
    for (i = 0; i < size; i++) { // i - номер прохода
        for (j = size - 1; j > i; j--) {
        // внутренний цикл прохода
            if (a[j - 1] > a[j]) {
                x = a[j - 1];
                a[j - 1] = a[j];
                a[j] = x;
            }
        }
    }
}

int BinarySearch (int A[], int Lb, int Ub, int Key) {
    int M;
    while(1){
        M = (Lb + Ub)/2;
        if (Key < A[M])
            Ub = M - 1;
        else if (Key > A[M])
            Lb = M + 1;
        else
            return M;
    if (Lb > Ub)
        return -1;
    }
}
```

```
int main(){
    srand(time(NULL));
    const long SIZE=10;
    int ar[SIZE];
    int key,ind;
    // до сортировки
    for(int i=0;i<SIZE;i++){
        ar[i]=rand()%100;
        cout<<ar[i]<<"\t";
    }
    cout<<"\n\n";
    /*
      Вызовем алгоритм пузырьковой сортировки,
      написанный нами ранее.
      Перед двоичным поиском данные нужно отсортировать.
    */
    bubbleSort(ar,10);

    // после сортировки
    for(int i=0;i<SIZE;i++){
        cout<<ar[i]<<"\t";
    }

    cout<<"\n\n";
    cout<<"Enter any digit:";
    cin>>key;
    ind=BinarySearch(ar,0,SIZE,key);
    cout<<"Index - "<<ind<<"\t"; cout<<"\n\n";
    return 0;
}
```

Двоичный поиск — очень мощный метод. Посудите сами: например, длина массива равна 1023, после первого сравнения область сужается до 11 элементов, а после второй — до 255. Легко посчитать, что для поиска в массиве из 1023 элементов достаточно 10 сравнений.

Функция, принимающая неограниченное количество параметров

Мы уже умеем успешно создавать функции, передавать параметры внутрь функций, возвращать значение из них.

Давайте подумаем можем ли мы решать такую задачу: требуется создать функцию, принимающую неограниченное количество параметров?

Например, мы хотим создать функцию, которая находит сумму всех переданных параметров. Количество параметров изначально неизвестно.

Реализовать такую задачу с учетом наших текущих знаний сложно. Нам всегда нужно указывать то количество параметров, которое было предусмотрено для вызова при определении функции. Можно попробовать решить эту задачу, передавая в качестве параметра массив. Это поможет нам не фиксироваться на количестве значений. Однако, в этом случае есть две проблемы:

1. Все параметры нужно собирать в массиве перед вызовом функции.
2. Если нам необходимо передать параметры разных типов, подход с массивом не подходит, так как массив содержит элементы одного тип.

Можно продолжать фантазировать, как решать эту проблему с помощью массива указателей, но лучше использовать готовое решение, которое существует в C++.

Для создания функции с неограниченным количеством параметров, необходимо воспользоваться макросами va_start, va_end, va_arg. Эти макросы находятся в библиотеке cstdarg.

Для создания функции с переменным количеством параметров нужно следовать таким правилам

1. Функция должна иметь один или более известных параметров. Эти известные параметры следуют перед списком переменных параметров. Под известными параметрами мы подразумеваем, явно заданные параметры при описании функции
2. Для старта прогулки по списку переменных параметров нужно вызвать va_start
3. Для получения параметра из списка параметров нужно вызывать va_arg. Обычно va_arg вызывается в цикле до тех пор, пока параметры не закончатся.
4. После получения всех параметров нужно вызвать va_end для корректного завершения работы

Рассмотрим каждый из макросов отдельно.

```
void va_start (va_list ap, lastParam)
```

Этот макрос инициализирует аргумент ap информацией для прохода по списку переменных параметров, lastParam — имя последнего известного параметра.

```
type va_arg (va_list ap, type)
```

Макрос va_arg вызывается для получения следующего значения из списка переменных параметров. В ap указывается переменная, инициализированная в va_start. Параметр type это ожидаемый тип следующего значения. Полученное значение возвращается из макроса va_arg. После вызова va_arg параметр ap изменяет своё значение,

чтобы предоставить доступ к следующему значению после полученного.

```
void va_end (va_list ap)
```

Этот макрос нужно вызвать, когда мы получили все требуемые значения. Он используется для корректного завершения работы. Аргумент ap – это параметр, инициализированный в va_start.

Рассмотрим использование этих макросов на примере.

Разработаем функцию, которая высчитывает минимум из полученных аргументов. Количество значений изначально неизвестно.

```
#include<iostream>
#include<cstdarg>
using namespace std;

/*
    Для указания того, что функция принимает
    неограниченное количество параметров нужно
    указать ...
    У нашей функции один известный параметр это
    numOfArgs. Этот параметр содержит число,
    переданных переменных параметров
*/
int GetMin(int numOfArgs,...){
    int minVal;
    va_list va;
    // инициализируем va для прохода по списку параметров
    // numOfArgs - самый правый известный параметр
    va_start(va,numOfArgs);
    // получаем первое значение из списка
    minVal = va_arg(va,int);
    int tempVal = 0;
```

```
    /*
      Мы получили уже первое значение из списка.
      Именно поэтому мы ведем цикл до numOfArgs - 1
    */
    for(int i = 0; i < numOfArgs - 1; i++){
        // В цикле получаем одно значение за другим
        tempVal = va_arg(va,int);
        if(minVal>tempVal)
            minVal =  tempVal;
    }
    // корректное завершение работы с переменными списка
    va_end(va);
    return minVal;
}

int main(){
    int res = GetMin(5, 10, -33, -1, 4, 9);
    cout<<"Minimum is: "<<res;
    return 0;
}
```

Из примера выше, вы узнали о том, что для обозначения переменного количества параметров используется «…».

Для прохода по списку используются уже знакомые вам макросы. Обязательно закрепите полученные вами знания на практике. Для этого решите задачу по подсчету среднеарифметического для переданных параметров.

5. Домашнее задание

1. Легенда гласит, что где-то в Ханое находится храм, в котором размещена следующая конструкция: на основании укреплены 3 алмазных стержня, на которые при сотворении мира Брахма нанизал 64 золотых диска с отверстием посередине, причем внизу оказался самый большой диск, на нем — чуть меньший и так далее, пока на верхушке пирамиды не оказался самый маленький диск. Жрецы храма обязаны перекладывать диски по следующим правилам:
   - За один ход можно перенести только один диск.
   - Нельзя класть больший диск на меньший.

   Руководствуясь этими нехитрыми правилами, жрецы должны перенести исходную пирамиду с 1-го стержня на 3-й. Как только они справятся с этим заданием, наступит конец света.
   Мы предлагаем Вам в качестве домашнего задания — решить данную задачу с помощью рекурсии. Желаем удачи!
2. Написать рекурсивную функцию нахождения степени числа.
3. Написать рекурсивную функцию, которая выводит N звезд в ряд, число N задает пользователь. Проиллюстрируйте работу функции примером.
4. Написать рекурсивную функцию, которая вычисляет сумму всех чисел в диапазоне от a до b. Пользователь

вводит a и b. Проиллюстрируйте работу функции примером.

5. Напишите рекурсивную функцию, которая принимает одномерный массив из 100 целых чисел, заполненных случайным образом, и находит позицию, с которой начинается последовательность из 10 чисел, сумма которых минимальна.