Указатели на функции

Назначение указателей на функции

Функция - это не переменная, но она имеет физическое положение в памяти (адрес), адрес может быть значением указателя (переменной адресного типа).

Адрес функции является входной точкой в функцию. Указатель может использоваться вместо имени функции.

Адрес функции получается при использовании имени функции без каких-либо скобок или аргументов (похоже на массивы).

Указатели на функции позволяют упростить решение многих задач путем создания функций высших порядков (т. е. функций, параметрами которых являются функции - «метафункций»).

Использование указателей на функции упрощает наращивание возможностей программы.

Использование указателей на функции

Пусть есть несколько функций с одним набором параметров, но выполняющих различные операции.

Тогда, если нужно изменить правила обработки данных, в «метафункции» можно указать только ссылку на нужную функцию (имя), не переписывая вызов целиком (см. пример lect-06-01.c)

Определим три функции:

```
long fact(unsigned n);
long qube(unsigned n);
long fibon(unsigned n);
```

И функцию **convol()** (от convolution - «свёртка»)

```
void convol(unsigned n, long(*funcName)(unsigned), long *res);
```

Использование указателей на функции

В функции **convol()** одним из параметров является имя функции, осуществляющей вычисления.

Функция возвращает значение через параметр **res**

```
void convol(unsigned n, long (*funcName)(unsigned), long *res)
{

    *res=funcName(n);
}
```

Использование указателей на функции

```
int main()
{
    unsigned value;
    long result;

    printf("Please enter value (less than 20!): ");
    scanf("%d", &value);
/* We need only set 2nd argument */
    convol(value,fibon,&result);

    printf("Result: %ld\n", result);
    return 0;
}
```

Можно написать **несколько функций** для вычисления, главное, чтобы **совпадали типы всех аргументов и результатов**.

Массив указателей на функции

Поскольку есть несколько однотипных функций, можно создать массив указателей на функции, тогда выбор функции можно оформить через ввод переменной (номера функции) и выбор элемента массива по номеру (индексу) (пример lect-06-02.c).

Объявления и реализации — те же, что и в предыдущем случае.

В `main()`:

```
long (*func[3])(unsigned); /* описание массива */
…
func[0]=fact;
func[1]=qube;
func[2]=fibon;
```

Массив указателей на функции

В main():

```
do
{
  CLS;
  printf("Functions to calculate:\n");
  printf("1 - Factorial\n");
  printf("2 - Cube of value\n");
  printf("3 - Fibonacci value\n");
  printf("Enter your choice: ");

   scanf("%d",&option);
} while((option<1)||(option>3));
…
convol(value,func[option-1],&result);
```

Динамический массив указателей на функции

Массив указателей на функции можно сделать динамическим.

Это полезно, если есть расширяемая библиотека функций (**.h**-файл и соответствующий **.c**-файл).

Тогда можно посчитать количество прототипов в **.h**-файле (**m**) и как-то получить их имена (обработать строки объявления прототипов).

Для динамического массива указателей на функции тоже требуется очистка памяти (пример lect-06-03.c).

В **main()**:

```
long (**func)(unsigned); /* описание массива */
…
m=3; /* нужно откуда-то получить количество элементов! */
func=(long(**)(unsigned))malloc(m*sizeof(long(*)(unsigned)));
/* проверка на NULL – обязательна! */
```

Динамический массив указателей на функции

Если **ввести новый тип** — указатель на вычисляющую функцию (через `typedef`), то код становится компактнее (пример lect-06-04.c).

```
typedef long(*calc)(unsigned);
void convol(unsigned n, calc, long *res);
```

В `main()`:

```
calc *func; /*  в блоке описаний */
…
m=3;
func=(calc*)malloc(m*sizeof(calc));
```

Рекурсивные функции.

Понятие «рекурсии»

1. Рекурсия — способ общего определения объекта или действия через себя, с использованием ранее заданных частных определений. Рекурсия используется, когда ***можно выделить самоподобие задачи***.

2. Рекурсия – способ организации вычислительного процесса, при котором подпрограмма в ходе выполнения составляющих ее операторов обращается сама к себе.

Чтобы такое обращение не было бесконечным, в подпрограмме должно быть ***условие, по достижению которого дальнейшего обращения не происходит***.

Рекурсивное обращение может включаться только в одну из ветвей подпрограммы.

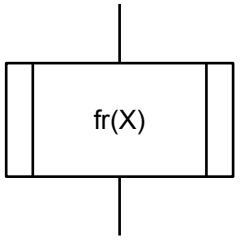
Варианты рекурсии в функциях

Прямая (простая, непосредственная) рекурсия – если в теле функции прямо содержится вызов этой же функции (функция непосредственно вызывает саму себя).

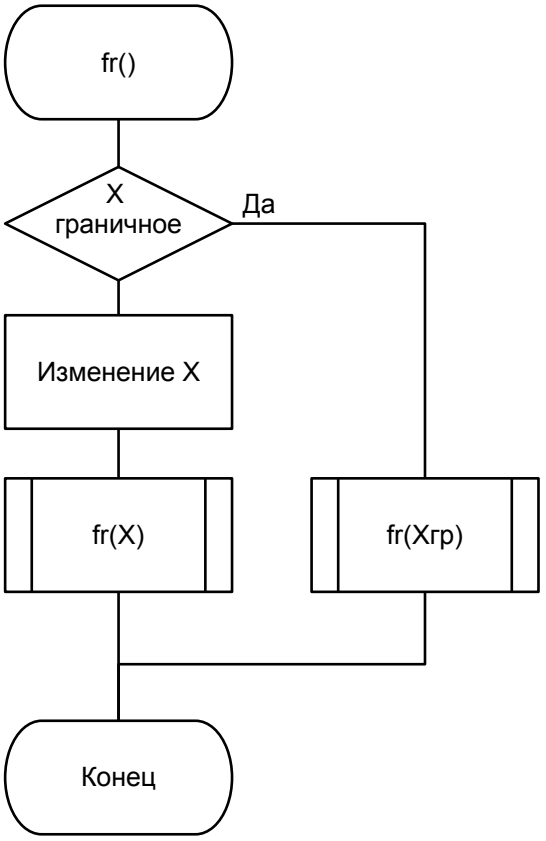
Косвенная (сложная) рекурсия – если в теле функции содержится вызов другой функции, которая, в свою очередь использует первую функцию (непосредственно или в теле еще одной вложенной функции).

Рекурсивные алгоритмы

Любой рекурсивный алгоритм должен предусматривать наряду с ***рекурсивным случаем*** еще и ***граничный случай,*** при реализации которого вычисления завершаются.



Условие полного окончания работы рекурсивной функции должно находиться в самой функции (иначе произойдет зацикливание), а именно, любая рекурсивная функция должна содержать рекурсивный вызов внутри условного оператора.



Рекурсия и итерации

1. Любой рекурсивный алгоритм можно преобразовать в эквивалентный итеративный (то есть использующий циклические конструкции).

2. Применение рекурсии в программе связано с рекурсивной природой математического описания алгоритма (если правила работы заданы рекурсивно, их имеет смысл реализовать рекурсивно).

3. Рекурсия связана с многократными вызовами процедур, а это при выполнении менее эффективно по сравнению с использованием циклов. Однако рекурсивные версии программ могут быть короче и нагляднее.

Примеры рекурсивных реализаций функций

1. «Классический» пример рекурсивного алгоритма — вычисление факториала

Определение (мат. формула): **N!=N*(N-1)!**

Код (приведенный в соответствие со структурной парадигмой, пример lect-06-05.c):

```
long factorial(int n)
{
    long f;
    if (n<0) f=0; /* to prevent input errors */
    else if(n<=1) f=1; /* condition to stop calculations */
    else f=n*factorial(n-1); /* recursive call */
    return f;
}
```

Примеры рекурсивных реализаций функций

2. Вычисление целой степени вещественного числа (пример lect-06-06.c)

```
double stepen(double a, int n)
{
    double st;
    if(n==0) st=1; /* condition to stop calculations */
    else if(n<0) st=1/stepen(a,-n);
    else st=a*stepen(a,n-1);
    return st;
}
```

Самостоятельно:

1. разобрать, что будет происходить при a=2, n=-3

2. записать алгоритм итерационно

Примеры рекурсивных реализаций функций

3. Вычисление числа Фибоначчи № n

(счет от 0 - пример lect-06-07.c)

```
long fibon(unsigned n)
{
    long f;
/* condition to stop calculations */
    if((n==0)||(n==1)) f=n;
/* recursive call */
    else f=fibon(n-1)+fibon(n-2);
    return f;
}
```

Рекурсивная сортировка

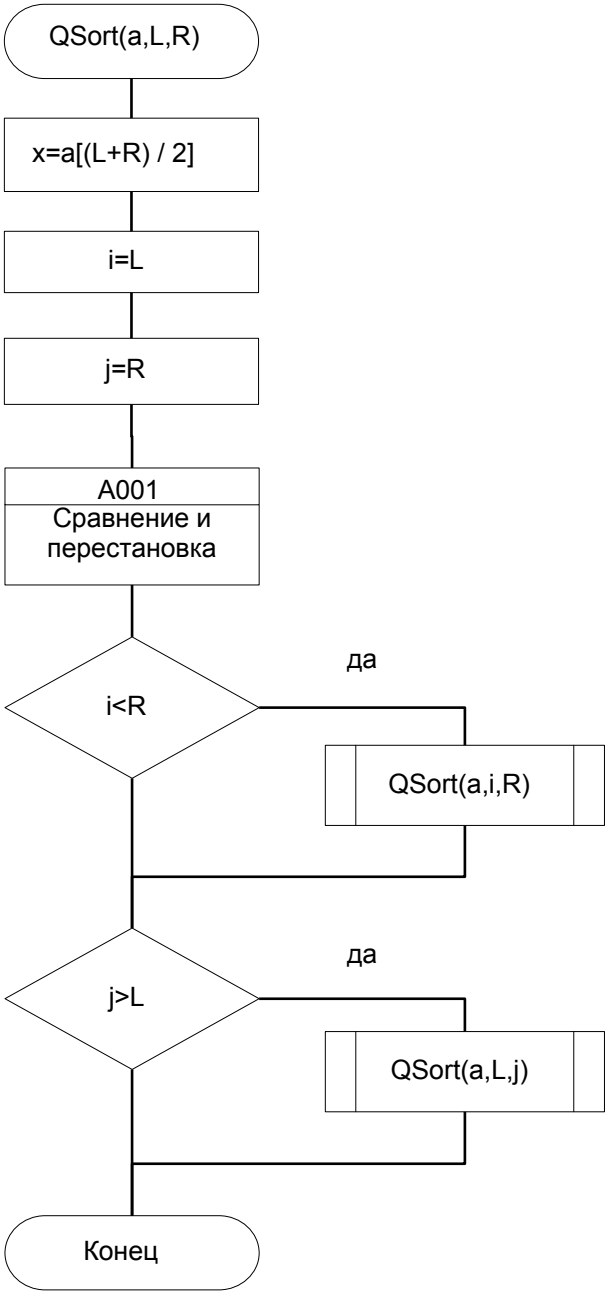
В **stdlib.h** определена функция сортировки **qsort()** (алгоритм Хоара).
Вычислительная сложность алгоритма в среднем (**N*log$_2$(N)**) операций (**N** — размер массива), в худшем случае — **N^2** операций (как в любых других методах сортировки).

Принцип работы (при сортировке по возрастанию):

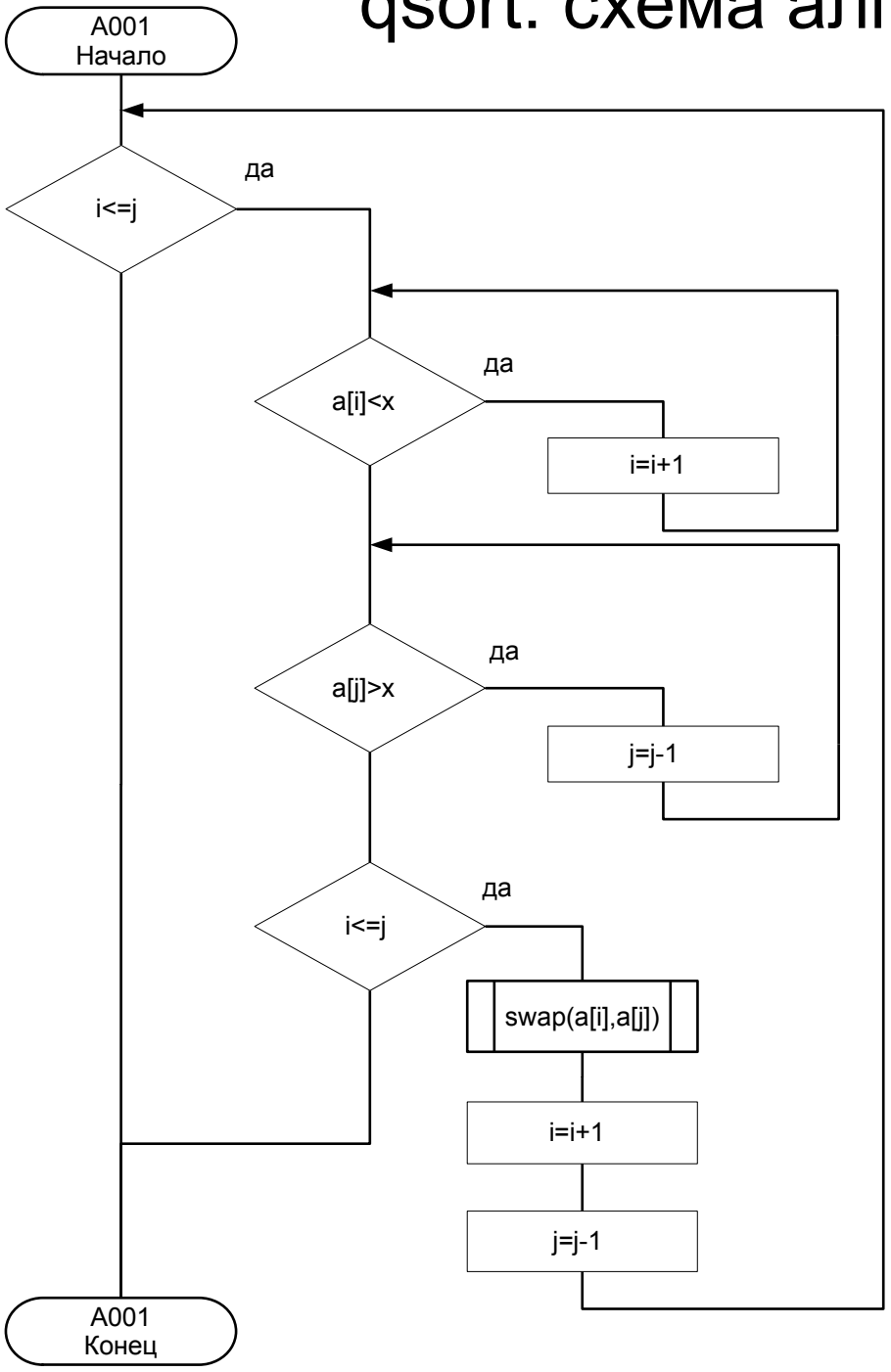
- Выбирается так называемый опорный элемент (как правило, средний элемент массива)
- Запускается процедура разделения массива, которая перемещает все элементы, меньшие, либо равные опорного, влево от него, а большие, либо равные опорному - вправо,
- В результате массив делится на два подмножества, причем значения в левом меньше либо равны значениям в правом подмножестве.
- Эти операции рекурсивно производятся с полученными подмассивами, если в них более двух элементов.

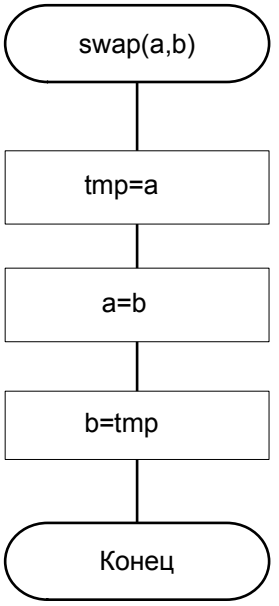
(подробности: https://ru.wikipedia.org/wiki/Быстрая_сортировка)

qsort: схема алгоритма



qsort: схема алгоритма





Рекурсивная сортировка

Пример реализации (**my_qsort()**) – пример lect-06-08.c

```
void my_qsort(int *arr,int b,int e)
{
    int left,right,middle;

    left=b;
    right=e;
 /* It's a reference element */
    middle=arr[(left + right)/2];
    while(left<=right)
    {
        while(arr[left]<middle) left++;
        while(arr[right]>middle) right--;
        if(left<=right) swap(&arr[left++],&arr[right--]);
    }
    if(b<right) my_qsort(arr,b,right);
    if(e>left) my_qsort(arr,left,e);
}
/* my_qsort (arr,0, n-1);  - так вызывается в main() */
```

Рекурсивная сортировка

Функция **swap()** для целых чисел:

```
void swap(int *a, int *b)
{
    int c;
    c=*a;
    *a=*b;
    *b=c;
}
```

В тех же адресах меняются значения.

Рекурсивная сортировка

Реальное использование **qsort()** (пример lect-06-09.c)

```
a=int_array(m)
puts("Initial array:");
for(i=0;i<m;i++) printf("%3d ", a[i]);

qsort(a,m,sizeof(int),(int(*)(const void*,const void*))cmp);

puts("\nSorted array:");
for(i=0;i<m;i++) printf("%3d ",a[i]);
```

Функция **cmp()** сравнивает два элемента. Результат — целое число (положительное, отрицательное или ноль).