acm.mipt.ru

олимпиады по программированию на Физтехе

Раздел «Язык Си» . CoffeeQsort :

Вступление

Как читать этот раздел?

Если вы в первый раз пытаетесь разобраться, как используется функция qsort, начните с разбора "Как написать универсальную функцию суммирования элементов массива".

Если вам нужно освежить память - идите сразу на "Использование qsort" и, если забыли, что такое указатель на функцию, то на "Указатель на функцию".

Постановка задачи

qsort - универсальная функция сортировки массива любого типа, на котором для двух его элементов задана функция сравнения этих элементов. Чтобы понять, как она вообще может работать, попробуем написать сами аналогичную функцию. Для простоты, пусть эта функция находит сумму элементов массива.

Как написать универсальную функцию суммирования элементов массива?

Попробуем написать функции суммирования n элементов массива для массива из int и массива из char. Сравним полученный результат и попробуем выделить общую часть.

Для каждого типа напишем свою функцию

```
#include <stdio.h>
int sum1 (int a[], int n) {
    int i, x;
    for(i=0, x=0; i<n; i++)
        x = x+a[i];
    return x;
}
int sum2 (char a[], int n) {
    int i, x;
    for(i=0, x=0; i<n; i++)
        x = x+a[i];
    return x;
}
int main()
{
    int  a[] = {7, -3, -12, 92, -4};
    char b[] = {7, -3, -12, 92, -4};
    printf("res=%d\n", sum1(a, 5));   // res=80
    printf("res=%d\n", sum2(b, 5));   // res=80
    printf("res=%d\n", sum2(a, 5));   // res=4 (используем неправильную функцию, тут ругается компилятор)

    return 0;
}
```

Заметим, что когда мы суммируем массив из int'ов с помощью функции, которая суммирует массив из char'ов, то получаем другой результат. Подумайте почему.

Функции очень похожи, но при попытке просуммировать один массив функцией для массива другого типа, появляются проблемы. На них нам указали при компиляции.

```
In function ‘main’:
20:5: warning: passing argument 1 of ‘sum2’ from incompatible pointer type
8:5: note: expected ‘char *’ but argument is of type ‘int *’
```

Заменим индекс на указатель

Заменим индекс i на указатель p и перепишем обе функции.

```
#include <stdio.h>
int sum1 (int * a, int n) {
    int x;
    int * p;
    for(p=a, x=0; p-a < n; p++)
        x = x + *p;
    return x;
}
int sum2 (char * a, int n) {
    int x;
    char * p;
    for(p=a, x=0; p-a < n; p++)
        x = x + *p;
    return x;
}
int main()
{
```

```
    int  a[] = {7, -3, -12, 92, -4};
    char b[] = {7, -3, -12, 92, -4};
    printf("res=%d\n", sum1(a, 5));
    printf("res=%d\n", sum2(b, 5));
    printf("res=%d\n", sum2(a, 5));

    return 0;
}
```

Различие только в типе передаваемого массива и типе указателя. Попробуем написать функцию sum, которая работает с универсальными указателями типа void *

```
int sum (void * a, int n) {
    int x;
    void * p;
    for(p=a, x=0; p-a < n; p++)  // на эту строку ругается компилятор
        x = x + *p;              // на эту строку ругается компилятор
    return x;
}
```

Действительно, примерять операцию * к переменной типа void * нехорошо. Ибо получаем результат типа void (что совсем плохо).

Кроме того, p++ для p разных типов будет отрабатываться по-разному. Так как в адресной арифметике увеличивается адрес в общем случае не на "1 байт", а на "1 элемент", то в случае, если p типа char*, размер 1 элемента будет ровно 1 байт. А в случае int *, адрес увеличится на sizeof(int) байт.

В случае void * адресная арифметика вообще даст undefined beheviour. Так что сведем всю адресную арифметику к типу, который работает заведомо с байтами.

И будем передавать в функцию **еще один параметр - размер одного элемента в байтах.** Тогда строка вызова этой функции будет sum(a, 5, sizeof(int))

```
int sum (void * a, int n, int k) {
    int x;
    char * p;
    for(p=a, x=0; p-a < n*k; p = p+k)
        x = x + *p; // что? мы берем только один байт как значение? а если там int?
    return x;
}
```

Для решения проблемы с применением * к типу void* сложнее. Если элементы массива a, адрес которого передан в параметрах, фактически имеют тип int, то для доступа к содержимому элемента массива по адресу p, нужно разыменовывать значение адреса, приведенное к нужному типу:
x = x + ***(int*)**p;
А если массив, на элементы которого указывает переменная p, имеет тип char, то
x = x + ***(char*)**p;

Давайте заменим этот *p на вызов функции get(p), типа int get(void * p), которая по адресу возвращает значение, которое *действительно* там лежит.

Для массивов разных типов придется использовать разные функции.

```
int get_int(void * p) {
    int x = * (int*) p;
    return x;
}
int get_char(void * p) {
    char x = * (char*) p;
    return x;
}
```

Во второй функции значение типа char автоматически преобразуется к типу int при вызове return x;.

Вопрос: как для разных массивов вызвать разные варианты функции get? Воспользуемся указателями на функции.

Указатель на функцию

Пусть у нас есть две функции, которые округляют дробное число до целого разными методами, например, foo1 и foo2. Их реализация:

```
int foo1(float);   // прототип функции foo1
int foo1(float x) {
    int y = (int)x;
    return y;
}
int foo2(float x) {
    int y = (int)(x+1);
    return y;
}
```

Так как функции тоже расположены где-то в памяти компьютера, введем понятие адреса начала функции и введем тип "указатель на функцию".

Сравните, в чем разница и что общего в оформлении прототипа функции и определении типа "указатель на функцию".

```
int foo1(float);     // прототип функции foo1
int (*xxx)(float);   // переменная xxx имеет тип "указатель на функцию, которая возвращает int и имеет единственный параметр типа
```

Использование указателей на функцию:

```
int main()
{
    int (*func_pointer)(float);     // завели переменную с именем func_pointer (могли придумать другое имя, хоть qqq, хоть my_libe
    func_pointer = foo1;            // в эту переменную записали указатель на функцию foo1
    int z = foo1(3.14);             // вызвали функцию foo1 с аргументом 3.14
    z = func_pointer(3.14);         // вызвали функцию foo1 с аргументом 3.14 (через указатель)
    func_pointer = foo2;            // в эту переменную записали указатель на функцию foo2
    z = foo2(3.14);                 // вызвали функцию foo2 с аргументом 3.14
    z = func_pointer(3.14);         // вызвали функцию foo2 с аргументом 3.14 (через указатель)
    return 0;
}
```

Универсальная функция сложения элементов массива. Окончательный вариант.

В универсальной функции суммирования добавим еще один аргумент - указатель на функцию get(p), типа int get(void * p), которая по адресу возвращает значение, которое *действительно* там лежит.

```
#include <stdio.h>
int get_int (void * p) {      // реализуем функцию, которая по адресу находит лежащий int
    int x = *(int*)p;
    return x;
}
int get_char (void * p) {     // реализуем функцию, которая по адресу находит лежащий char
    char x = *(char*)p;
    return x;
}
int sum (void * a, int n, int k,
        int(*get)(void*)     // указатель на функцию, которую мы будем передавать в аргументе get
        ) {
    int x;
    void * p;
    for(p=a, x=0; p-a<n*k; p = p+k)
        x = x + get(p);      // вызов функции, адрес которой лежит в аргументе get
    return x;
}
int main()
{
    int  a[] = {-7, 3, -12, 92, 4};
    char b[] = {-7, 3, -12, 92, 4};
    printf("res=%d\n", sum(a, 5, sizeof(int),  get_int));    // передача имени функции в виде аргумента
    printf("res=%d\n", sum(b, 5, sizeof(char), get_char));   // передача имени функции в виде аргумента

    return 0;
}
```

qsort - стандартная функция языка C для сортировки массива

В языке C, как и во многих других языках программирования, реализована стандартная функция сортировки массива.

Описание функции qsort и ее аргументы

man 3 qsort

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
int(*compar)(const void *, const void *));
```

Функция **qsort** упорядочивает массив из *nmemb* элементов размером *size*. Аргумент *base* указывает на начало массива. Содержимое массива располагается по возрастающему принципу, согласно функции сравнения, указанной в параметре *compar* и имеющей два аргумента (адреса сравниваемых элементов массива).

Функция сравнения должна возвращать целое число

- равное **0**, если значение, на которое указывает первый аргумент, **равно** значению, на который указывает второй элемент;
- **< 0**, если значение, на которое указывает первый аргумент, **меньше** значения, на который указывает второй элемент;
- **> 0**, если значение, на которое указывает первый аргумент, **больше** значения, на который указывает второй элемент;

Если два члена массива равны, то порядок их расположения в массиве не определен.

Заметим, что функция qsort имеет аналогичные аргументы, как наша универсальная функция суммирования:

- адрес начала сортируемого массива
- количество сортируемых элементов
- размер одного элемента в байтах
- указатель на функцию (но не получения значения по адресу, а сравнение двух элементов, которые лежат по этим адресам).

Использование функции qsort для сортировки одного массива int и второго массива char

```
#include <stdio.h>
#include <stdlib.h> // чтобы работал qsort
void print(int a[], int n);
void print_char(char a[], int n);

// 0 if x==y
// <0 if x<y
// >0 if x>y
// определение функции сравнения для массива int'ов
int cmp_int(const void * p1, const void * p2) {
    int x = *(int *)p1; // добываем из указателя значение по этому указателю
    int y = *(int *)p2; // добываем из указателя значение по этому указателю
    return x-y;
}
// определение функции сравнения для массива char'ов
int cmp_char(const void * p1, const void * p2) {
    char x = *(char *)p1; // добываем из указателя значение по этому указателю
    char y = *(char *)p2; // добываем из указателя значение по этому указателю
    return x-y;
}
int main()
{
    int a[] = {-7, 3, -12, 92, 4};
    char b[] = {-7, 3, -12, 92, 4};

    print(a, 5);
    qsort(a, 5, sizeof(int), cmp_int);  // сортировка массива a
    print(a, 5);

    print_char(b, 5);
    qsort(b, 5, sizeof(char), cmp_char);// сортировка массива b
    print_char(b, 5);

    return 0;
}
void print(int a[], int n) {
```

```
    int i;
    for (i = 0; i<n; i++)
        printf("%d ", a[i]);
    printf("\n");
}
void print_char(char a[], int n) {
    int i;
    for (i = 0; i<n; i++)
        printf("%d ", (int)a[i]);
    printf("\n");
}
```

О сортировке структур

Пример из книги Ворожцова/Винокурова

Задачи (можно сдать в проверяющей системе)

- Реализуйте функцию int cmp_int(const void * p1, const void * p2); сравнения двух int'ов.
- Отсортируйте последовательность целых чисел по возрастанию.
- Отсортируйте последовательность целых чисел по убыванию.
- Отсортируйте последовательность дробных чисел по возрастанию.
  - обязательно введите в последовательность, отличные менее, чем на 1 числа.
- Сортировка символов в строке по возрастанию ASCII кодов.
- Сортировка последовательности натуральных чисел по единицам, потом по десяткам, потом по сотням...
- Сортировка только нечетных, четные числа остаются на своих местах.
  - используйте дополнительную память.
- Сортировка строк
  - Знание strcmp и fgets облегчит жизнь.
- сортировка символов в строке по заданному алфавиту по возрастанию (строка только из символов алфавита)
  - Знание strchr облегчит жизнь.
- Реализуйте функцию int cmp_Point(const void * p1, const void * p2) для сравнения структур
- отсортируйте точки по удаленности от центра координат
- отсортируйте студентов по номеру группы и фамилии.