

## Лекция 4. Обзор языка С, продолжение

### Обработка последовательности

Первая серия учебных задач связана с обработкой последовательностей, и мы уже готовы к тому, чтобы программировать решение подобных задач.

#### Абстрактная постановка.

Есть набор чисел неопределенной длины. Надо вычислить некоторую характеристику этого набора чисел за один просмотр последовательности, т.е. нельзя числа сохранять, нельзя возвращаться назад, нельзя читать повторно. После того как очередное число из последовательности обработано, оно становится более недоступно.

Математический алгоритм решения подобных задач опирается на идею доказательства по индукции (есть даже некоторая теория индуктивных алгоритмов). Другими словами, решение задач часто сводится к подбору некоторых значений и рекуррентных соотношений для них, по которым эти значения можно пересчитывать шаг за шагом. Естественно, нужно также определиться с начальными условиями, с которых начинается расчет.

В качестве простейшего примера можно рассмотреть задачу вычисления среднего арифметического последовательности чисел.

Подойдем с точки зрения индукции.

Пусть мы знаем значение  $S$  среднего арифметического для некоторого уже просмотренного участка последовательности. И только что получили очередное число  $x$  из последовательности. Можем ли мы вычислить среднее арифметическое для нового, удлиненного участка последовательности? Ответ, естественно, нет. Но если мы вместе со средним арифметическим знаем количество прочитанных ранее чисел  $n$ , то пересчет среднего арифметического (и нового количества элементов) выполняется очевидным образом:

$$S = (S*n + x)/(n+1);$$
$$n = n + 1;$$

Можно сделать еще проще, если вычислять просто сумму чисел  $A$  и их количество  $n$ :

$$A = A + x;$$
$$n = n + 1;$$
$$S = A/n; \quad (\text{только когда будет надо})$$

Теперь у нас все есть, чтобы запрограммировать решение этой задачи — формулы для вычислений и примеры последовательного чтения данных из файла.

#### Пример: sequence-mean.cpp

Другие задачи решаются аналогично. Выбирается некоторый набор характеристик, которые можно пересчитывать друг за другом при продвижении по последовательности, и через которые можно выразить требуемый ответ. (более подробно — на семинарах).

## Указатели

В языке C есть еще один тип данных — указатель (pointer).

Смысл — адрес размещения объекта в памяти.

Типизированный указатель — адрес с учетом размера объекта

```
int *p, *q;
double *r;
const char *s;
```

нетипизированный указатель — просто абстрактный адрес

```
void *a;          --- можно потом привести к конкретному типу
```

Операции с указателями:

сравнения == !=

сложение вычитание с целым —  $p+1$   $q-10$   $p+k$  — сдвиг по памяти кратно размеру объекта

вычитание указателей —  $p - q$  можно (один тип) - количество объектов

$p - r$  нельзя (разные типы)

приведение типа — `(char *)p` !!! если понимаешь зачем !!!

разименование — `*p` — доступ к значению по адресу памяти

операция `&` — взятие адреса (указателя) от объекта

операция `*` — разименование указателя — доступ к значению по данному адресу

```
int x;
int *p;
p = &x;
x = 25;
```

\*p есть 25

```
*p = 321;
```

x есть 321 (и \*p, естественно, тоже)

чему равно `*(p+1)` ??? (а это как повезет ...)

можно ли писать ???

```
p = 12345;
```

формально да, но смысла нет ...

можно писать

```
p = 0;      NULL    nullptr
```

Зачем это все надо?

1. работа с параметрами функций через указатель
2. работа с массивами

Помните? : при вызове функции параметры передаются в функцию по значению (т.е. копируются) Поэтому функция принципиально не может изменить значения своих фактических параметров.

```
void f1(int x)                void f2(int *p)
{
    x = 100;                  {
}                               *p = 100;
                               }
```

```
int x=1, y=2, z=3;
int *p, *q;
f1(x);
f1(y);
f1(z);
значения x,y,z остались прежними (1,2,3)
p = &x;
q = &y;
f2(p);
f2(q);
f2(&z);
значения x,y,z стали 100
значения p,q не изменились
```

Параметр-указатель дает функции возможность работать непосредственно по указанному адресу памяти, т.е. именно в том месте, где размещена переменная (объект).

При вызове копируется только значение параметра-указателя (адрес), что играет роль при объемных параметрах (в C такое нечасто, но в C++ сплошь и рядом)

Итак, полезно и удобно для работы с функциями, когда надо получить несколько значений в результате.

Работа с массивами

```
int a[10];
int *p;
p = a;      // имя массива есть указатель на его начальный элемент
p = &a[0];  // это то же самое
p+1 есть указатель на a[1], ... p+k --- на a[k]

a[k]      *(a+k)
```

Задача: в массиве заведомо есть нулевое значение, надо найти его позицию (индекс)

```
int a[100], n = 100, i;
int *p;
...
for (i=0; i<n; i++) {
    if (a[i] == 0) break;
}
i есть искомый индекс
```

```
for (p=a; *p; p++);    // но если 0 нет, то будет плохо ...
i = p - a;
i  есть искомый индекс
```

Передача массива в функцию.

Задача: найти максимум и минимум элементов массива

```
void MinMax (int n, double *x, double *pmin, double *pmax)
{
    int i;
    *pmin = *pmax = x[0];
    for (i = 1; i<n; i++)
    {
        if ( *pmin > x[i]) { *pmin = x[i]; }
        if ( *pmax < x[i]) { *pmax = x[i]; }
    }
}
.....
double zzz[1024], zmin, zmax;
.....
MinMax (1024, zzz, &zmin, &zmax);           // scanf("%lf",&zmax);
```

А что будет для таких вызовов ?

```
MinMax (128, zzz, &zmin, &zmax);
MinMax (2048, zzz, &zmin, &zmax);
MinMax (100, zzz + 200, &zmin, &zmax);
MinMax (1022, zzz + 2, zzz, zzz + 1);
```