Московский государственный университет
физический факультет
кафедра квантовой теории и физики высоких энергий

В.А.Ильина П.К.Силаев

Краткий справочник по языку “C”

Москва 2012

Содержание

1 Введение

Это ни в коем случае не учебник “C”, а именно шпаргалка, которая дает возможность писать элементарные программы. Незнание любого приведенного здесь факта делает программирование на “C” практически невозможным.

Шпаргалка эта не претендует на особенную аккуратность. Главное предупреждение, которое необходимо сделать, касается приведенных в тексте примеров. В них часто перемешаны описания переменных и исполняемые операторы. В настоящей программе описания должны идти до исполняемых операторов.

Сделаем несколько замечаний общего характера о счетных программах.

Во-первых, касательно стиля программирования. Вы должны понимать, что хороший стиль программирования не есть инвариантная величина. То, что является обязательным требованием для кусочка какого-либо большого проекта (front end к какой-нибудь базе данных), становится совершенно необязательным, а зачастую и вредным, если Вы пишете счетную программу для себя. Нет ничего страшного в использовании глобальных переменных, если Вы пишете не кусочек очень большого проекта. И стандартная рекомендация: “если функции для работы нужна память, то в начале работы функции она должна эту память заказать, а в конце — освободить” является не просто бесполезной, а вредной, потому что сильно замедляет работу программы (если функция вызывается достаточно часто). Гораздо разумнее застолбить эту память навсегда — глобально или статически.

Во-вторых, касательно процедуры написания и отладки счетной программы. Стандартный **абсолютно неверный** подход заключается в том, что Вы пишете здоровенную программу от начала до конца, а потом начинаете ее компилировать и запускать на счет. После этого как правило произносятся фразы: “я все написал правильно, а машина не работает”. Или даже: “это ошибка в компиляторе”. Все это проявление мании величия средней степени. Разумеется, надо поступать совершенно иначе. Надо написать маленький кусочек программы, и проверить, как он работает. (При этом следует всегда помнить, что 100% гарантии тут нет — даже если этот кусочек правильно работает при тех входных данных, которые Вы использовали при тестировании, он может начать работать неправильно при других входных данных. Как правило такую ошибку поймать труднее всего). Дальше следует дописать еще кусочек, и тоже его оттестировать. И так далее.

2 Минимальные требования к счетной программе на “C”

1. Программа не должна запрашивать входные данные и параметры с клавиатуры и не должна включать в себя входные данные. Входные параметры могут вводиться с командной строки (либо из конфигурационного файла), а входные данные должны считываться из файла данных.

2. Программа, если она считает дольше десятка секунд, должна выводить на экран минимальную информацию о текущем этапе вычислений.

3. Результаты счета должны выводиться в файл результатов, а не только на экран.

4. Программа должна быть легко читаема. Не надо бояться лишних пробельных строк и закомментированных строк-разделителей.

5. Не следует гоняться за лаконичностью в ущерб простоте — Ваша задача состоит в написании понятной, работающей и легко отлаживаемой программы, а не в демострации гибкости, которую допускает язык “C”.

6. В программе не должны встречаться загадочные числа 7, 666 и т.п. Все эти константы должны быть заменены посредством `#define` на подходящие имена. Как правило, эти имена пишутся большими буквами, чтобы отличить константы от настоящих переменных.

7. Однобуквенные имена переменных допустимы для переменных цикла или вспомогательных переменных, но отнюдь не для переменных, несущих смысловую нагрузку — имена этих переменных должны быть осмысленными.

8. Не следует все операции запихивать в функцию `main()`. Крупные логические блоки непременно следует выделять в функции. С другой стороны, не следует дробить программу на кучу маленьких функций, каждая из которых вызывает следующую, которая, в свою очередь, вызывает следующую и т.д.

9. Совершенно необязательно все данные, которые получает и возвращает функция, запихивать в ее аргументы. Выбор между использованием глобальной переменной, использованием формального параметра, использованием адреса переменной как формального параметра и использованием оператора `return` должен делаться в зависимости от конкретной ситуации.

3 Минимальные сведения о типах переменных языка "C"

1. Во-первых, существуют переменные типа `char`, т.е. буквы. Как правило, эти переменные имеют размер в 1 байт (т.е. 8 бит) и меняются в пределах $-128 \ldots 127$. Если отрицательные значения нежелательны, можно использовать модификатор `unsigned`: переменные `unsigned char` меняются в пределах $0 \ldots 255$. В счетных программах переменные `char` употребляются главным образом для сооружения строк (см. ниже). Чтобы присвоить переменной `char` значение, соответствующее коду какой-либо буквы, достаточно написать

   ```
   buk='D';
   ```

   На машине с ASCII-кодировкой такая запись вполне эквивалентна загадочной и неразумной записи "`buk=68;`".

   Кроме "обычных" (печатаемых) букв существуют и спецсимволы: `'\n'` — переход на новую строку; `'\t'` — табуляция; `'\r'` — переход на первую позицию текущей строки; `'\a'` — "звонок", т.е. одиночный писк терминала; `'\''` — двойная кавычка; `'\' '` — одиночная кавычка; `'\\'` — сам символ "\" (обратный слэш).

2. Во-вторых, существуют переменные типа `int`, т.е. целые числа. Допустимы модификаторы `unsigned` и `long`: "`long int`", "`unsigned int`" и "`unsigned long int`". Диапазон изменения этих переменных зависит и от типа машины и от компилятора. Обычно `int` — 2 байта (т.е. $-32768 \ldots 32767$ и $0 \ldots 65536$ для `unsigned int`), а `long int` — 4 байта (т.е. $-2147483648 \ldots 2147483647$ и $0 \ldots 4294967296$ для `unsigned long int`). Нередко оба типа — `int` и `long int` по 4 байта.

3. В-третьих, это переменные типа `double`. Форма их записи традиционна для алгоритмических языков: `1.2345`, `-0.7777777777`, `-33.4444444e-86`, `-2.1111e33` (Существует еще тип `float`, но переменные этого типа безусловно не должны встречаться в счетных программах. Экономия памяти, которую они дают, довольно иллюзорна, а потеря точности будет очень существенна. Кроме того, использование `float` как правило приводит к замедлению работы программы.)

Тип любой используемой переменной должен быть определен. Это делается так:

```
char buk;
int i,j,k;
double energy;
```

Подробнее о области действия этих определений см. ниже. Кроме того, Вам необходимо знать о модификаторе "`static`" (также см. ниже).

4 Массивы и указатели в языке "C"

Обычно в счетных программах употребляются одномерные и двумерные массивы. Массив фиксированного размера определяется так:

```
int numer[9],indeks[20][30];
double coord[7];
```

Число в квадратных скобках задает размер массива. После этих определений можно ссылаться на любой элемент массива

```
indeks[5][27]=555;
coord[3]=coord[0];
```

Нумерация элементов массива начинается с нуля. Поэтому последний (девятый) элемент массива `numer` — это `numer[8]`. Попытка написать `numer[9]` — постоянная ошибка. Компиляторы языка "C" **не проверяют** неправильность индекса (отрицательный индекс или индекс, больший или равный размеру массива). Более того, если Вы положили число по неверному адресу, Вы с некоторой вероятностью его оттуда и возьмете. Поэтому неверная программа может иногда делать вид, что она работает правильно. Неустойчивость работы программы (разные результаты при последовательных запусках) — верный признак неправильной работы с памятью.

В том случае, когда размер массива заранее неизвестен, приходится прибегать к динамической аллокации памяти.

Для одномерных массивов следует сначала определить указатель

```
double *vect;
```

а затем, когда уже стал известен размер массива (например, 95), отвести память под массив и поместить указатель на его начало:

```
vect=(double *)malloc(95*sizeof(double));
if(vect==NULL) {printf(''No mem for vect!\n''); exit(1); }
```

Функция `malloc` ничего не знает о типах переменных, поэтому она измеряет память просто в байтах и возвращает указатель типа "`void *`". Так что размер массива в байтах вычисляется с помощью функции `sizeof`, которая возвращает размер переменной типа `double` в байтах, а запись "`(double *)`" является принудительным преобразованием типа от "`void *`" к "`double *`".

Кстати, совершенно аналогично можно принудительно преобразовывать и другие типы. Например, целочисленной переменной можно присвоить значение типа `double`: "`n=1.5;`", но лучше записать это как "`n=(int)1.5;`". Более того, "`3/2`" равно единице, а вот "`3/2.0`" или "`3/(double)2`" равно 1.5.

Если памяти не хватает, функция `malloc` возвращает константу `NULL`. В этом случае необходимо немедленно прекращать работу программы с помощью функции `exit`. Чтобы пользоваться функциями `malloc` и `exit`, в заголовке программы необходимо поставить

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

Для двумерных массивов следует сначала определить указатель на массив указателей:

```
int **dat;
```

когда размер массива уже известен (например, 20×30), надо сначала отвести память под массив указателей:

```
dat=(int **)malloc(20*sizeof(int *));
if(dat==NULL) {printf(''No mem for dat!\n''); exit(1); }
```

А затем для каждого из 20 указателей провести такую процедуру: отвести память под 30 целых чисел и поставить этот указатель на начало соответствующего массива:

```
for(i=0;i<20;i++)
   {dat[i]=(int *)malloc(30*sizeof(int));
    if(dat[i]==NULL) {printf(''No mem for dat!\n''); exit(1); }}
```

Если диапазон изменения первого индекса ($0 \ldots 19$) известен заранее, то аллокацию памяти под массив указателей можно опустить, т.к. можно сразу описать массив фиксированного размера из 20 указателей:

```
int *dat[20];
```

Если памяти хватило, то динамически аллоцированным массивом можно пользоваться точно так же, как и массивом фиксированного размера.

Следует понимать, что указатель — это попросту переменная, содержащая адрес того или иного участка памяти. Тип указателя "`int *`" или "`double *`" позволяет компилятору правильно сдвигаться в памяти при изменении индекса массива. (Для массива `double` изменение индекса на 1 — это сдвиг на 8 байт, а для массива `long int` — сдвиг на 4 байта). В принципе указатель можно ставить и не на начало массива:

```
int *uk,ind[21],num;
uk=&(ind[10]);
```

Операция `&` — это (помимо всего прочего) еще и операция взятия адреса объекта. В результате указатель `uk` выставлен на середину массива `ind` и законные значения индексов для него — это $-10 \ldots 10$, причем `uk[-10]` — это синоним `ind[0]`, `uk[0]` $\equiv$ `ind[10]`, `uk[10]` $\equiv$ `ind[20]`.

Более того, можно даже написать

```
uk=&num;
```

Тогда единственным законным значением индекса будет "0", и `uk[0]` — это то же самое, что `num`.

Язык "C" достаточно гибок и позволяет программисту определять свои собственные сложные типы данных. Однако, в счетной программе эти возможности вряд ли полезны, так что структуры, объединения и битовые поля мы обсуждать не будем.

5 Минимальные сведения о ядре языка "C"

Как ни странно, набор операторов языка "C" отнюдь не удовлетворяет требованиям минимальности, поэтому ряд операторов в хорошей счетной программе использоваться не должен. В частности, операторы `switch` и `goto` безусловно не должны появляться в счетных программах.

5.1 Арифметические операторы и операторы присваивания

Существуют 4 стандартные арифметические операции:

`x=3+2;`	$\Longrightarrow$	x=5	`x=3+2.0;`	$\Longrightarrow$	x=5.0
`x=3-2;`	$\Longrightarrow$	x=1	`x=3.0-2;`	$\Longrightarrow$	x=1.0
`x=3*2;`	$\Longrightarrow$	x=6	`x=3.0*2;`	$\Longrightarrow$	x=6.0
`x=3/2;`	$\Longrightarrow$	x=1	`x=3/2.0;`	$\Longrightarrow$	x=1.5

То обстоятельство, что "`3/2`" равно единице, служит **постоянным источником ошибок**.

Стандартной операции возведения в степень не существует.

Существует оператор остатка при целочисленном делении

`x=85%20;` $\Longrightarrow$ x=5

`x=80%20;` $\Longrightarrow$ x=0 (нулевой остаток)

`x=99%20;` $\Longrightarrow$ x=19 (максимальный остаток)

Существуют операции модификации

`x=7; x++;`	$\Longrightarrow$	x=8	`x=7; x--;`	$\Longrightarrow$	x=6
`x=7; x+=3;`	$\Longrightarrow$	x=10	`x=7; x-=3;`	$\Longrightarrow$	x=4
`x=7; x*=3;`	$\Longrightarrow$	x=21	`x=33; x/=3;`	$\Longrightarrow$	x=11

Операторы модификации не только экономят время при наборе программы, но и делают ее читабельнее.

5.2 Логические операторы

Эти операторы имеют значения "истина" (true) и "ложь" (false). При этом числа также имеют логическое значение: ноль (0) — это false, а любое ненулевое число — это true. Поэтому вполне допустимы конструкции

```
if(i%2) printf(''Nechetnoe''); else printf(''Chetnoe'');
```

Лучше все же писать

```
if(i%2!=0) printf(''Nechetnoe''); else printf(''Chetnoe'');
```

Существуют стандартные операторы сравнения

`>`	больше	`>=`	больше или равно
`<`	меньше	`<=`	меньше или равно
`!=`	не равно	`==`	равно

Последнее соглашение служит постоянным источником ошибок. Желание писать `if(x=3)` совершенно неистребимо. Эта запись синтаксически вполне законна: она означает, что переменной "x" присваивается значение "3", сам оператор присвоения возвращает значение "3", т.е. условный оператор проверяет истинность числа "3", которое отлично от нуля, т.е. это всегда истинное условие. Аналогично, `if(x=0)` — это всегда ложное условие.

Существуют логические операторы для конструирования составных условий

`&&`	"и" (конъюнкция)
`||`	"или" (дизъюнкция)
`!`	"не" (отрицание)

5.3 Побитовые операторы

В программах для численного счета эти операторы встречаются сравнительно редко, разве что при реализации алгоритма FFT (быстрого преобразования Фурье). Следует понимать, что результат побитовой операции зависит от типа ее аргумента. Мы проиллюстрируем их на примере переменных типа `unsigned char` размером в 1 байт.

```
unsigned char a,b,c;
```

`a=12;`	$\Longrightarrow$	`0000 1100`	
`b=10;`	$\Longrightarrow$	`0000 1010`	
`c=a&b;`	$\Longrightarrow$	`0000 1000`	c=8, побитовое "и"
`c=a|b;`	$\Longrightarrow$	`0000 1110`	c=14, побитовое "или"
`c=!b;`	$\Longrightarrow$	`1111 0101`	c=245, побитовое "не"
`c=a<<1;`	$\Longrightarrow$	`0001 1000`	c=24=2*a, побитовый сдвиг влево на 1 позицию
`c=a>>1;`	$\Longrightarrow$	`0000 0110`	c=6=a/2, побитовый сдвиг вправо на 1 позицию
`c=b<<3;`	$\Longrightarrow$	`0101 0000`	c=80=8*b, побитовый сдвиг влево на 3 позиции

`c=b<<6;`	$\Longrightarrow$	`1000 0000`	c=128, побитовый сдвиг влево на 6 позиций

У переменных со знаком старший бит является знаковым битом. Поэтому

```
char buk;
long int num;
buk=127; buk++;
num=2147483647 ; num++;
```

приведет к результату `buk`$= -128$ и `num`$= -2147483648$, что служит **неизменным источником ошибок**.

5.4 Условный оператор

Допустимые варианты синтаксиса иллюстрируются примерами

```
if(x>3) {a=7; b=8; c=9; } else {a=9; b=8; c=7; }
```

```
if(x>3) {a=7; b=8; c=9; }
```

```
if(x>3) a=7; else a=9;
```

```
if(x>3) ; else a=7;
```

```
if(x>3) a=7;
```

Результат действия оператора `if` очевиден — при выполнении условия первый оператор или первый набор операторов, ограниченный фигурными скобками (блок) исполняется, в противном случае исполняется оператор или блок, расположенный после ключевого слова `else`. Вторая половина условного оператора (начиная с `else`) может быть опущена.

Существуют несколько характерных ошибок при написании условного оператора. Одна из них уже упоминалась:

```
x=5; if(x=3) a=7; else a=9;
```

Здесь нет синтаксической ошибки, но `a` будет равно 7.

```
x=1; a=9; if(x>3); a=7;
```

Здесь тоже нет синтаксической ошибки, но `a` будет равно 7.

К счастью, другие аналогичные ошибки вызывают ругань компилятора:

```
if(x>3) {a=7; b=8; c=9; };   else {a=9; b=8; c=7; };
```

Можно порекомендовать всегда ставить фигурные скобки в условном операторе, даже вокруг одного оператора. Это несколько уменьшает читабельность, но и уменьшает вероятность ошибки при редактировании (если вдруг понадобится дописать еще пару операторов).

5.5 Оператор цикла `for`

Допустимые варианты синтаксиса иллюстрируются такими примерами:

```
double x[100];
for(i=0;i<100;i++)
   { x[i]=20.0-0.666*i;
    if(i%10==0) x[i]+=7.0; }
```

Это идиоматическое выражение при заполнении массива. Допустимые значения индекса `i` в массиве равны $i = 0, \ldots 99$, поэтому переменная цикла `i` инициализируется нулем, на каждом шаге цикла она увеличивается на единицу, а цикл выполняется, пока справедливо условие `i<100`, т.е. вплоть до `i=99`.

Порядок, в котором проводится инициализация, проверяется условие, проводится модификация переменной цикла и выполняется тело цикла, поясняется таким примером: для

```
for(i=0 ; i<0 ; i++) {тело цикла }
```

тело цикла не выполнится ни разу; для

```
for(i=0 ; i<1 ; i++) {тело цикла }
```

тело цикла выполнится ровно 1 раз при значении i=0; для

```
for(i=0 ; i<3 ; i++) {тело цикла }
```

тело цикла выполнится 3 раза при значениях i=0,1,2.

Следующий пример

```
a=0;
for(i=3 ; i<100 && 3*i<290 ; i=i+17+i%3) a+=i;
```

Здесь иллюстрируется возможность сложного условия, возможность нетривиального закона изменения переменной цикла и то, что фигурные скобки вокруг тела цикла могут быть опущены.

```
i=5; a=0;
for(;;)
   {if(i>=1000) break;
    i=i*(i-2);
    if(i%7<3 && i<250) continue;
    a+=i; }
```

Это также идиоматическое выражение — цикл без переменной цикла, без инициализации, условия цикла и закона модификации переменной цикла. Следует заметить, что любой из этих трех элементов может быть опущен по отдельности, т.е. вполне допустимо

```
i=0; for(;i<5;i++) ...
```

либо

```
for(i=0;;) { if(i>=5) break; i++; }
```

и т.п. Однако чаще всего все же употребляется `for`-цикл, в котором опущено все сразу. Управление таким циклом реализуется условными операторами в сочетании с ключевыми словами `continue` и `break`.

Оператор `continue`, который может появиться в любом цикле, вызывает переход на следующий виток цикла. Это чрезвычайно полезно в тех ситуациях, когда внутри цикла есть ветвящееся дерево возможностей, и при некоторых условиях анализировать дальнейшие возможности уже не нужно, можно идти дальше. Разумеется, посредством большого числа вложенных условных операторов либо посредством введения сложных условий в условные операторы всегда можно достичь желаемого эффекта, но оператор `continue` позволяет написать то же самое гораздо короче.

Оператор `break` аналогичен оператору `continue`, только он не запускает новый виток, а обрывает выполнение цикла (т.е. реализует выход из цикла). Ясно, что без этого оператора опускать условие цикла было бы нельзя — цикл просто никогда не кончился бы.

Оба оператора, разумеется, влияют только на один цикл, а именно на тот, в котором они находятся. Например для двух циклов по i и по j

```
for(i=0;i<100;i++)
   { for(j=99;j>=0;j-)
      {if(i%10==0) continue;
       if(j<10) break; a+=j; }
   a=a+7; }
```

операторы `continue` и `break` относятся только к циклу по j (При `i` кратных 10 переменная `j` пробегает все значения от 99 до 0, но переменная `a` не изменяется, а при остальных значениях `i` переменная `j` пробегает от 99 до 9, причем вклад в величину `a` дают только значения от 99 до 10. Цикл по `i` операторов `break` и `continue` не замечает. Более того, наивная попытка написать `break; break;` (один `break` для цикла по `j`, второй — для цикла по `i`) ни к чему не приведет.

Характерные ошибки при написании циклов похожи на ошибки для условного оператора.

```
for(i=0;i<100;i++); { a+=9; b*=a; c=i; a=b+c; }
```

Здесь нет синтаксической ошибки, но "тело цикла" (точнее, то что автор этой программы считает телом цикла) исполнится ровно 1 раз, да притом при значении i=100.

```
for(i=0;i<100;i++)
   { for(j=0;j<50;i++) { a+=i+j; }}
```

Здесь тоже нет синтаксической ошибки, но цикл по j никогда не кончится, при запуске этой программы возникнет (ложное) впечатление, что она "зависла".

В принципе существуют и другие типы циклов, но почти никаких новых возможностей они не добавляют, так что лучше ограничиться `for`-циклом. В качестве иллюстрации приведем `while`-цикл:

```
while(fscanf(fp,''%s'',str)!=EOF)
{ обработка строки str }
```

вполне эквивалентно

```
for(;;)
   { if(fscanf(fp,''%s'',str)==EOF) break;
   обработка строки str }
```

5.6 Определение функций

Как правило целесообразно выделять логические блоки программы в отдельные функции. При разумном подходе это делает программу лаконичнее и читабельнее. Синтаксис определения функции поясняется следующими примерами:

```
double devjat(double x, int n)
{
int i;
double tmp;
tmp=1.0;
for(i=0;i<n;i++) tmp*=x;
if(tmp>1.0e10) return 1.0e10;
if(tmp<-1.0e10) return -1.0e10;
return tmp;
}
```

Эта функция безграмотно и неэкономно возводит x в целочисленную степень n и обрезает ответ по абсолютно величине на 10^{10}. Тип функции (т.е. тип возвращаемой ею величины) и количество и тип ее аргументов определяются в заголовке. После заголовка идет описание локальных (внутренних) переменных функции, а далее исполняемые операторы.

```
void krik(int flag)
{
if(flag>0) printf(''Polozitelen\n'');
if(flag<0) printf(''Otritsatelen\n'');
}
```

Эта функция не возвращает никакого значения и просто печатает соответствующие сообщения при положительном или отрицательном значении аргумента `flag` (при нулевом значении она не делает ничего).

```
void vopl(void)
{
printf(''Eto function vopl\n'');
}
```

Эта функция не имеет аргументов.

5.7 Параметры командной строки

Параметры командной строки — это чрезвычайно эффективный способ передавать программе небольшое количество входных параметров (если параметров много, их разумнее поместить во входной файл). Доступ к параметрам командной строки дают аргументы функции "`main`" (см. ниже).

Обыкновенно эта функция определяется как

```
int main(void)
```

или даже как

```
void main(void)
```

Впрочем, некоторые компиляторы ругаются, если "main" определено не как "`int`". А другие ругаются, если она определена как "`int`", а оператора "`return`" в ней нет. Дело в том, что раньше хорошим тоном считалось возвращать системе код завершения программы. Обыкновенно "0" (ноль) считается признаком успешного завершения, а натуральное число — признаком ошибки той или иной степени тяжести. Вернуть системе код завершения можно двумя способами: или "`exit(7);`" — из любого места программы, или "`return 13;`" — из функции "`main`".[1]

Вернемся к параметрам командной строки. Если Вам нужны параметры командной строки, то функцию "`main`" следует определять иначе — с аргументами:

```
int main(int npar, char **par)
```

В этом случае программе станут доступны все параметры командной строки. Допустим, Вы запустили программу таким образом:

```
./myprog -w 1.2345 myout77
```

Тогда после запуска переменная "npar" примет значение "4" — это полное число параметров командной строки, а сами параметры будут находится в двумерном массиве "char", т.е. в одномерном массиве строк, в следующем виде:

- в `par[0]` будет "`./myprog`"
- в `par[1]` будет "`-w`"
- в `par[2]` будет "`1.2345`"
- в `par[3]` будет "`myout77`"

Тем самым `par[3]` — это готовое имя выходного файла, а числовой double-параметр "1.2345" легко извлечь из `par[2]` с помощью функции "`sscanf`" (см. ниже).

6 Минимальный набор библиотечных функций языка "C"

6.1 Математические функции.

Чтобы пользоваться математическими функциями, в заголовке программы необходимо поставить строку

```
#include <math.h>
```

Существуют тригонометрические и гиперболические функции

```
cos(x)   sin(x)   tan(x)   cosh(x)   sinh(x)   tanh(x)
```

Обратные тригонометрические и гиперболические функции

[1] Система помещает код завершения программы в какую-нибудь переменную среды (environment variable), например, в `PIPESTATUS`.

`acos(x)` `asin(x)` `atan(x)` `acosh(x)` `asinh(x)` `atanh(x)`

Экспонента, натуральный логарифм, корень квадратный и абсолютная величина

`exp(x)` `log(x)` `sqrt(x)`

Существует функции целой части и абсолютной величины

`floor(x)` $\Longrightarrow [x]$ `ceil(x)` $\Longrightarrow [x]+1$ `fabs(x)` $\Longrightarrow |x|$

`floor(5.9)`$= 5.0$ `ceil(5.9)`$= 6.0$

`floor(-5.9)`$= -6.0$ `ceil(-5.9)`$= -5.0$

(Следует понимать, что принудительное преобразование типа `double` в тип `int` дает результат, отличающийся от результата функций `floor` или `ceil`, именно, `(int)(5.9)`$= 5$, а `(int)(-5.9)`$= -5$).

Все вышеперечисленные функции возвращают `double` и аргументом имеют тоже `double`.

Существует функция возведения в степень

`pow(x,y)` $\Longrightarrow$ x^y

оба аргумента имеют тип `double`, возвращается также `double`.

Существует двухаргументный арктангенс

`atan2(y,x)` $\Longrightarrow$ $\mathrm{arctg}(y/x)$ с учетом знаков x и y:

`atan2( 1.0, 1.0)` $= \pi/4$ `atan2( 1.0,-1.0)` $= 3\pi/4$

`atan2(-1.0,-1.0)` $= -3\pi/4$ `atan2(-1.0, 1.0)` $= -\pi/4$

оба аргумента имеют тип `double`, возвращается также `double`.

Необходимо иметь в виду следующее. Хотя большинство компиляторов на IBM-совместимых машинах спокойно переносят выражения типа `sqrt(4);` или

```
int i; i=25; sqrt(i);
```

лучше заранее приучить себя не полагаться на это, поскольку при переносе на другую машину окажется, что `sqrt(4)=1.596e-138`, а `pow(2,3)=7.371e+069`.

Особенно много ошибок такого рода связано с функцией pow. Следует понимать, что `pow(4.0,3/2)` — это 4.0, а вовсе не 8.0, что функция `pow(x,2.5)` во много раз медленнее, чем `x*x*sqrt(x)`, а `y=pow(x,9.0)` не только гораздо медленнее чем

```
tmp=x*x; tmp=tmp*tmp; y=tmp*tmp*x;
```

но и не может быть использована при отрицательных x. (Функция `pow(x,y)` — это в сущности exp(y*ln(x))).

Существует целочисленная функция абсолютной величины

`abs(n)` $\Longrightarrow$ $|n|$

эта функция имеет аргумент `int` и возвращает также `int`.

6.2 Функции для работы со строками

Строка в языке “C” — это последовательность букв, завершающаяся признаком конца строки — символом `'\0'`, который (как правило) просто равен нулю. Разумеется, эта последовательность помещается в одномерном массиве типа `char`. Соответствие размера массива длине строки (строка вместе с символом `'\0'` должна умещаться в массиве) и само наличие символа `'\0'` в конце строки — на совести программиста. Например, если строковая переменная (массив типа char) определена как

```
char str[50];
```

то максимальная длина строки — 49.

Чтобы не было необходимости задавать строки побуквенно, определена конструкция

`''abc''` или `''012ASD345''` или даже

`''line N 1 \nline N 2 \a (bell)\n''`

— это строки-константы, снабженные нулевым символом в конце, которые можно подставлять всюду вместо строковых переменных.

Чтобы можно было пользоваться функциями для работы со строками, в заголовке программы следует поставить

```
#include <string.h>
```

Функция `strlen(str);` возвращает длину строки str, т.е. индекс элемента массива, в котором сидит `'\0'`. Наличие такого элемента в массиве — на совести программиста.

Функция `strcpy(str1,str2);` копирует содержимое строки str2 в строку str1. Например, после

```
strcpy(str,''Dfg'');
```

`str[0]` будет равно `'D'`, `str[1]` — `'f'`, `str[2]` — `'g'`, `str[3]` — `'\0'`, а прочие значения `str[i]` не изменятся.

Функция `strcat(str1,str2);` дописывает содержимое строки str2 в конец строки str1.

Функция `strchr(str,buk);` ищет первую (слева) букву `buk` в строке `str` и возвращает указатель на это место. Если такой буквы нет, возвращается `NULL`. Например, после

```
char str[50],*ukaz;
strcpy(str,''abcd1234'');
ukaz=strchr(str,'d');
```

`ukaz[0]` будет равно `'d'`, `ukaz[1]` — `'1'`, `ukaz[2]` — `'2'`, и т.д.

Функция `strstr(str1,str2);` ищет первое (слева) место в строке `str1`, где встречается строка `str2` и возвращает указатель на это место. Если строка `str2` нигде не встречается, возвращается `NULL`.

6.3 Функции ввода-вывода

Чтобы пользоваться функциями ввода-вывода, в заголовке программы необходимо поставить строку

```
#include <stdio.h>
```

Далее, ввод-вывод может идти по следующим каналам:

- В некоторый файл и из некоторого файла.
- По стандартым каналам ввода-вывода.
- Из строки и в строку.

В первом случае файл должен быть открыт для ввода или вывода. Это делается следующим образом

```
FILE *fp1,*fp2;
fp1=fopen(''myfile.txt'',''r'');
```

Первый аргумент функции `fopen` — имя файла. Можно явно указать путь к файлу

```
fp1=fopen(''/home/username/mydir/myfile.txt'',''r'');
```

Если путь к файлу не указан, то речь идет о файле в текущей директории. Второй аргумент функции `fopen` — тип доступа. Тип `''r''` — чтение файла в текстовом режиме. Тип `''w''` — запись файла в текстовом режиме, при этом если файл с указанным именем уже существует, то запись пойдет "поверх" него, от старого содержимого ничего не останется (даже если будет записан всего 1 байт). Тип `''a''` — запись файла в текстовом режиме, но "аддитивная" ("append"), т.е. если файл с указанным именем уже существует, то запись будет дописана в его конец, старое содержимое файла полностью сохранится. Перейти в режим двоичной записи или чтения позволяет модификатор `''b''`, т.е. `''rb''` — чтение файла в двоичном режиме; `''wb''` — запись файла в двоичном режиме; `''ab''` — добавление к файлу в двоичном режиме.

Например:

```
fp2=fopen(''myfile.txt'',''w'');
```

файл открывается для записи в текстовом режиме;

```
fp2=fopen(''myfile.txt'',''ab'');
```

файл открывается для добавления в двоичном режиме.

Двоичный и текстовый режимы отличаются следующим: признаком конца текстового файла является символ EOF (конец файла, ASCII код 26). Поэтому если Вы попытаетесь читать двоичный файл (а в двоичном файле наверняка рано или поздно встретится байт "26") и используете для этого текстовый режим, то при считывании символа EOF дальнейшее чтение файла прекратится.[2]

При попытке открыть несуществующий файл для чтения функция `fopen` возвращает `NULL`, так что обычно пользуются идиомой

```
fp=fopen(''myfil.dat'',''r'');
if(fp==NULL) {printf(''Input file not found!\n''); exit(1); }
```

Разумеется, открытый файл нужно потом закрыть:

```
fclose(fp1);
```

Форматный ввод из файла реализуется следующим образом:

```
int n;
fscanf(fp1,''%d'',&n);
```

это чтение целого числа в переменную `n`. Чтение идет из файла, который ранее был открыт для чтения. Второй аргумент функции — это форматная строка. Строка `''%d''` означает, что производится чтение целого числа типа `int`. Соответственно, `''%ld''` означает тип `long int`, `''%u''` — тип `unsigned`, `''%lu''` — тип `unsigned long`, `''%le''` — тип `double`, а `''%s''` — строковую переменную (при чтении строкой считается любая совокупность непробельных символов). При этом считываемые объекты отделяются друг от друга одним или более пробелами, символы табуляции и переходы на новую строку приравниваются к пробелам. Третий аргумент — это адрес переменной, в которую помещается результат считывания. Соответствие типа переменной типу, указанному в форматной строке, абсолютно необходимо. Чтение по формату `''%d''` в переменную типа `double` приводит к ошеломляющим результатам. Постоянная ошибка при употреблении `fscanf` — писание имени, а не адреса переменной. Если считывание производится не в переменную, а в элемент массива, то пишется адрес этого элемента, т.е. `&(dat[7][33])`. Наконец, для строковых переменных адресом является адрес начала строки. Однако, `&(str[0])` в действительности является синонимом `str`. Так что разумнее всего писать

```
char str[200];
fscanf(fp1,''%s'',str);
```

Следует иметь в виду, что многие компиляторы спокойно переваривают явно неверную запись " `fscanf(fp1,''%s'',&str);` ", но на других компиляторах это не пройдет, так что лучше сразу приучиться писать правильно.

Далее, если надо прочесть несколько переменных подряд, то совершенно необязательно читать их по одной. Допустим, в файле содержатся значения целого числа и некоторой вещественной величины в таком виде:

```
num1   -555      Xvalue[13]:
   1.2345e-66
```

Тогда разумнее всего прочитать их так:

```
char str1[200],str2[100];
```

[2]Еще одно отличие проявляется при работе под DOS/Windows: в DOS/Windows конец строки есть два символа подряд: CR (возврат каретки, ASCII код 13) и LF (новая строка, ASCII код 10). В результате в текстовом режиме при записи один символ `'\n'` преобразуется в два: CR и LF, а при чтении — наоборот, два символа CR и LF преобразуются в один `'\n'`. В двоичном режиме, напротив, никаких преобразований не производится. Кстати, в UNIX конец строки — это один символ LF, именно поэтому DOS'овские файлы в UNIX'овом редакторе выглядят странновато.

```
int num;
double x[40];
fscanf(fp1,''%s%d%s%le'',str1,&num,str2,&(x[7]));
```

Функция `fscanf` возвращает число успешно считанных полей (это удобно при отладке процесса чтения из файла). Если достигнут конец файла, она возвращает константу `EOF`. Поэтому в данном случае она возвратит число 4 и запихнет значение -555 в переменную `num`, а $1.2345 \cdot 10^{-66}$ в восьмой элемент массива `x`. Строковые переменные `str1` и `str2` будут содержать `''num1''` и `''Xvalue[13]:''`.

Фрагмент программы

```
for(;;) { if(fscanf(fp,''%d'',&nnn)==EOF) break; ... }
```

является идиомой для чтения из файла с неопределенным количеством целых чисел.

Форматный вывод в файл реализуется следующим образом:

```
int n;
fprintf(fp1,''Numer=%d\n'',n);
```

это печать в файл, открытый для записи или добавления, целого числа, содержащегося в переменной n. Вместо `''%d''` можно написать `''%3d''`, тогда число при выдаче будет занимать минимум 3 позиции, лишние позиции будут заняты пробелами (это удобно при печати чисел в столбик):

```
Numer=  1      при n=1
Numer=111      при n=111
Numer=11111    при n=11111
```

Совершенно аналогично, `''%s''` — это печать строковой переменной, а `''%14s''` — это печать строковой переменной в минимум 14 позиций, `''%ld''` — это печать переменной типа `long int`. Для переменных типа `double` есть еще один параметр — число цифр после запятой, так что возможны варианты: `''%le''` — печать с 6 цифрами после запятой, `''%23le''` — печать с 6 цифрами после запятой в минимум 23 позиции, `''%.14le''` — печать с 14 цифрами после запятой, `''%23.14le''` — печать с 14 цифрами после запятой в минимум 23 позиции.

Как и для `fscanf`, можно выводить по несколько переменных и по несколько строк за один раз:

```
fprintf(fp1,''1: N1=%d N2=%d\n2: X0=%.12le X5=%.12le \n\n'',
                                                    n1,n2,x[0],x[5]);
```

Ввод-вывод по стандартным каналам в точности аналогичен вводу-выводу в файл. В сущности, можно считать, что для каждой программы изначально открыты три файла — `stdin` (клавиатура), `stdout` (экран) и `stderr` (экран). При этом `fscanf(stdin,''%d'',&n)` есть синоним `scanf(''%d'',&n)`, а `fprintf(stdout,''%d'',n)` есть синоним `printf(''%d'',n)`. Разница между `stdout` и `stderr` заключается в буферизации вывода. Вывод по каналу `stderr` не буферизуется. А вот по каналу `stdout` бывает по-разному. У многих компиляторов функция `printf` не рисует выдачу на экране немедленно, а накапливает в буфере. Фактический вывод реализуется либо при переполнении буфера (скажем, больше 128 букв), либо при получении символа новой строки `'\n'`. Так что для трассировки приходится писать

```
if(i%100==0) {printf(''[%d]'',i/100); fflush(stdout); }
```

функция `fflush` вызывает принудительное опорожнение буфера.

Наконец, ввод-вывод из и в строку символов реализуется функциями

```
char str1[80],str2[160];
str1[0]=' '; str1[1]='-'; str1[2]='2'; str1[3]='5'; str1[4]=0;
sscanf(str1,''%d'',&n);
sprintf(str2,''Num=%6d'',n);
```

в результате число -25 запишется в переменную n, а строка str2 будет содержать текст

```
Num=   -25
```

7 Минимальные сведения о структуре программы на языке “C”

7.1 Функция “main” и другия функции

Во всякой программе должна присутствовать функция `main`. Исполнение программы начинается с исполнением этой функции, а завершение ее исполнения означает выход из программы.

Что касается расположения других функций, то в любом случае в тот момент, когда компилятор встречает вызов функции, он должен знать ее тип и количество и тип ее аргументов. Проще всего, если функция уже определена к этому моменту

```
double myfun(double x, int n)
{ return x+n; }

main()
{
printf(''%le'',myfun(5.0,3));
}
```

Это простейший и, пожалуй, самый разумный вариант. Бывает, что неудобно или невозможно определить функцию до того, как она встречается в программе. Тогда можно написать так

```
double myfun(double, int);

main()
{
printf(''%le'',myfun(5.0,3));
}

double myfun(double x, int n) { return x+n; }
```

Первая строка в этом примере задает прототип функции, и компилятору этого достаточно, чтобы правильно написать вызов функции.

Упоминавшиеся выше строки типа `#include <math.h>` нужны как раз для того, чтобы вставить в текст Вашей программы прототипы соответствующих стандартных функций (в данном случае — математических). Разумно делать это в самом начале программы, т.е. писать заголовок программы, состоящий из `#include ...`.

Минимальный заголовок:

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
```

Если используются строковые функции следует дописать:

```
#include<string.h>
```

Кроме прототипов функций в этих файлах определены некоторые полезные константы. Определение констант реализуется так:

```
#define NNN 200
```

После этого везде в программе имя `NNN` заменится на 200. В принципе директива `#define` позволяет определять и макросы, но использование макросов — это постоянный источник ошибок:

```
#define kub(x) x*x*x
```

определяет макрос, который как бы возводит в куб что угодно. Более того, `kub(3)` или `kub(a)` сработают правильно, а вот `kub(d+f)`, естественно, нет. **Не используйте макросы в счетных программах — это постоянный источник ошибок.**

Многие компиляторы терпеливо относятся к расположению директив `#include` и `#define`. Их можно писать хоть на середине строчки. Однако некоторые компиляторы полагают, что эти директивы обязаны стоять строго в начале строки (т.е. символ `#` должен быть первым символом строки).

7.2 Область локализации переменных

По области локализации и выделению памяти переменные делятся на локальные и глобальные. Кроме того, следует обсудить и формальные параметры функций.

Глобальные переменные определяются вне функций. Они доступны для любой функции в том смысле, что она может получать и изменять их значение.

Локальные переменные определяются внутри данной функции, они не видны из других функций. Более того, значения локальных нестатических переменных ("автоматических" переменных) по завершении работы функции теряются.

Предупреждение. Одной из самых типичных ошибок при использовании автоматических переменных является попытка наплодить этих переменных в количестве, скажем, аж 80 Kb. Автоматические переменные заводятся в стеке, а стек по умолчанию крайне невелик. Компилируете Вы с оптимизацией, и, следовательно, проверка переполнения стека из программы выкидывается, чтобы программа побыстрей работала. В результате Вы получаете непредсказуемое поведение программы. Вы можете завести десяток double-переменных, но если речь идет о локальном double-массиве из, скажем, 50 элементов, **всегда определяйте его как "`static`"**:

```
static double massiv[50];
```

В результате массив будет помещен не в стеке, а в области данных. (Грубо говоря, там же, где живут глобальные переменные). Другим отличием статических переменных от нестатических ("автоматических") является то, что их значения от вызова до вызова функции сохраняются (собственно, поэтому они и называются "статические"). Иногда это может быть очень удобно, а иногда это превращается в недостаток. Например, при рекурсивном вызове функций эти переменные использовать с очень большой осторожностью (ведь несколько разных вызовов одной функции будут обращаться к одному и тому же участку памяти). В сущности, статические переменные — это те же самые глобальные, только их не видно из других функций.

Как уже было сказано, описания локальных переменных должны предшествовать первому исполняемому оператору в функции.

Формальные параметры функций также локализованы в пределах данной функции.

Не надо бояться передавать данные с помощью глобальных переменных. Все утверждения о том, что "применение глобальных переменных — это плохой стиль программирования" относятся к работе над большими проектами, где действительно необходима максимальная локализация данных и структурированность программы. Ваша задача — написать читабельную, правильно и (по возможности) быстро работающую счетную программу, а не борьба за максимальную структурированность и инкапсулированность. Между тем бесконечные аллокации и деаллокации локальных данных, многочисленные вызовы функций с передачей большого количества данных могут сильно замедлить работу программы.

Если речь идет об одном большом массиве, с которым разные функции выполняют разные манипуляции, то логично не засовывать его в формальные параметры этих функций, а сделать его глобальным. То же самое относится к величине (типа `int` или `double`), общей для всей программы.

Однако, следует соблюдать осторожность: если в нескольких функциях употребляется переменная цикла i, то **она должна быть своей (локальной) для каждой из функций**. Попытка сэкономить на строках программы:

```
int i;

double sloz(double x)
{ double z;
z=0; for(i=0;i<7;i++) z+=x; return z; }

double umnoz(double x)
{ double z;
z=1; for(i=0;i<5;i++) z*=sloz(x); return z; }
```

очевидно, приведет к ошибке: цикл в функции `umnoz` оборвется уже на первом обороте.

Обсудим, наконец, вопрос о формальных параметрах функций. Если формальный параметр имеет тип `int` или `double`, то функции при вызове передается значение аргумента, так что вопрос о том, может ли функция

```
double vosem(double x) { x=x*x; x=x*x; x=x*x; return x; }
```

изменить значение переменной `x` при вызове `vosem(x);` столь же нелеп, как вопрос о том, не заменится ли внутри машины число 2 на число 256 при вызове `y=vosem(2);`.

Совершенно другая ситуация с массивами и указателями. В этом случае функции передается адрес массива или переменной и она вполне может пойти по этому адресу и изменить соответствующее значение. (Именно поэтому у функции `fscanf` третьим аргументом является не сама переменная, а ее адрес). Функция

```
void zamena(char *str)
{
int i;
for(i=0;i<strlen(str);i++) if(str[i]=='d') str[i]='F';
}
```

заменит в любой строке все буквы `'d'` на буквы `'F'`. Аналогично, программа

```
void tri(double *x, int n)
{ int i; for(i=0;i<n;i++) x[i]=3.0*i; }

main()
{ double dat[20];
tri(dat,10);
printf('"%le\n",dat[3]);
}
```

действительно заполнит первые 10 элементов массива соответствующими значениями и напечатает число 9.0.

Единственная тонкость касается двумерных массивов.

Если определена функция с прототипом

```
void myfun(int **n,double **x);
```

то динамически аллоцированные массивы (определенные как "`int **ind;`" и "`double *datt[20];`"), могут появиться в аргументах этой функции в виде

```
myfun(ind,datt);
```

а двумерные массивы фиксированного размера (определенные как "`int indd[10][20];`" и "`double dat[20][40];`"), по крайней мере для большинства компиляторов — не могут. Конечно, двумерные массивы фиксированного размера тоже можно запихнуть в аргументы функций, но мы этого обсуждать не будем, проще сделать эти массивы глобальными.

Простейшая и, пожалуй, самая разумная структура счетной программы такова: сначала идет заголовок с директивами `#include`, затем определяются константы с помощью директив

`#define`, затем идут описания всех глобальных переменных, далее определяются функции, по возможности так, чтобы определение функции шло до ссылки на нее в другой функции, далее функция `main`. В функции `main` логично начать с выяснения размера и аллокации всех динамически аллоцируемых массивов (разумеется, после описания всех локальных переменных).

7.3 Пример простейшей бессмысленной программы

Приведем простейший и довольно бессмысленный пример. Эта программа зачитывает из входного файла вектор неизвестной длины n, действует на него довольно странной прямоугольной матрицей $10 \times n$ и выводит ответ в выходной файл. Формат входного файла:

```
n 3
x1 1.0
x2 2.0
x3 3.0
```

Программа проводит минимальную проверку входного файла.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define INFILE  "myprog.in"
#define OUTFILE "myprog.out"

#define NUM  10
#define FC   0.123

int nnn;
double *invect,*matr[NUM];
double  outvect[NUM];

void matr_x_vect(void)
{
int i,j;

for(i=0;i<NUM;i++)
  {outvect[i]=0.0;
   for(j=0;j<nnn;j++) {outvect[i]+=matr[i][j]*invect[j]; }   }
}

void main(void)
{
int i,j;
FILE *fp;
char tmpstr[80];

fp=fopen(INFILE,"r");
if(fp==NULL) { printf("Input file %s not found!",INFILE); exit(1); }
```

```
if(fscanf(fp,"%s%d",tmpstr,&nnn)!=2)
       { printf("Error reading n!"); exit(1); }

invect=(double *)malloc(nnn*sizeof(double));
if(invect==NULL)
       { printf("Not enough memory for invect!"); exit(1); }
for(i=0;i<NUM;i++)
  {matr[i]=(double *)malloc(nnn*sizeof(double));
    if(matr[i]==NULL)
         { printf("Not enough memory for m(%d)!",i+1); exit(1); }
   }

for(i=0;i<nnn;i++)
{
if(fscanf(fp,"%s%le", tmpstr, &(invect[i]) )!=2)
       { printf("Error reading x%d!",i+1); exit(1); }
}
fclose(fp);

for(i=0;i<NUM;i++)
{
for(j=0;j<nnn;j++)
  {
    matr[i][j]=exp(i*0.7-j*1.0e-4);
    if( i%3==0 && (j%2!=0 || j<5) ) { matr[i][j]*=sin(FC*(i-j));  }
    else        {  matr[i][j]*=cos(FC*(i+j)); matr[i][j]-=33.3*i; }
  }
}

matr_x_vect();

fp=fopen(OUTFILE,"w");
for(i=0;i<NUM;i++)
  { fprintf(fp,"outvect[%3d] = %14.4le\n",i+1,outvect[i]); }
fclose(fp);

}
```