Глава 1. Обзор языка

Начнем с быстрого ознакомления с языком Си. Наша цель – показать на реальных программах существенные элементы языка, не вдаваясь в мелкие детали, формальные правила и исключения из них. Поэтому мы не стремимся к полноте и даже точности (заботясь, однако, о корректности примеров). Нам бы хотелось как можно скорее подвести вас к моменту, когда вы сможете писать полезные программы. Чтобы сделать это, мы должны сконцентрировать внимание на основах: переменных и константах, арифметике, управлении последовательностью вычислений, функциях и простейшем вводе-выводе. В настоящей главе мы умышленно не затрагиваем тех средств языка, которые важны при написании больших программ: указателей, структур, большой части богатого набора операторов, некоторых управляющих инструкций и стандартной библиотеки.

Такой подход имеет свои недостатки. Наиболее существенный из них состоит в том, что отдельное характерное свойство языка не описывается полностью в одном месте, и подобная лаконичность при обучении может привести к неправильному восприятию некоторых положений. В силу ограниченного характера подачи материала в примерах не используется вся мощь языка, и потому они не столь кратки и элегантны, как могли бы быть. Мы попытались по возможности смягчить эти эффекты, но считаем необходимым предупредить о них. Другой недостаток заключается в том, что в последующих главах какие-то моменты нам придется повторить. Мы надеемся, что польза от повторений превысит вызываемое ими раздражение.

В любом случае опытный программист должен суметь экстраполировать материал данной главы на свои программистские нужды. Новичкам же рекомендуем дополнить ее чтение написанием собственных маленьких программ. И те и другие наши читатели могут рассматривать эту главу как “каркас”, на который далее, начиная с главы 2, будут “навешиваться” элементы языка.

1.1 Начнем, пожалуй

Единственный способ выучить новый язык программирования – это писать на нем программы. При изучении любого языка первой, как правило, предлагают написать приблизительно следующую программу:

Напечатать слова Hello, world

Вот первое препятствие, и чтобы его преодолеть, вы должны суметь где-то создать текст программы, успешно его скомпилировать, загрузить, запустить на выполнение и разобраться, куда будет отправлен результат. Как

только вы овладеете этим, все остальное окажется относительно просто. Си-программа, печатающая “Hello, world”, выглядит так:

```
#include <stdio.h>
main()
{
    printf("Hello, world\n");
}
```

Как запустить эту программу, зависит от системы, которую вы используете. Так, в операционной системе UNIX необходимо сформировать исходную программу в файле с именем, заканчивающимся символами ".c", например в файле hello.c, который затем компилируется с помощью команды

```
cc hello.c
```

Если вы все сделали правильно – не пропустили где-либо знака и не допустили орфографических ошибок, то компиляция пройдет “молча” и вы получите файл, готовый к исполнению и названный a.out. Если вы теперь запустите этот файл на выполнение командой

```
a.out
```

программа напечатает

```
Hello, world
```

В других системах правила запуска программы на выполнение могут быть иными; чтобы узнать о них, поговорите со специалистами.

Теперь поясним некоторые моменты, касающиеся самой программы. Программа на Си, каких бы размеров она ни была, состоит из *функций и переменных*. Функции содержат *инструкции*, описывающие вычисления, которые необходимо выполнить, а переменные хранят значения, используемые в процессе этих вычислений. Функции в Си похожи на подпрограммы и функции Фортрана или на процедуры и функции Паскаля. Приведенная программа – это функция с именем *main*. Обычно вы вольны придумывать любые имена для своих функций, но “main" – особое имя: любая программа начинает свои вычисления с первой инструкции функции *main*.

Обычно *main* для выполнения своей работы пользуется услугами других функций; одни из них пишутся самим программистом, а другие берутся готовыми из имеющихся в его распоряжении библиотек. Первая строка программы:

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

сообщает компилятору, что он должен включить информацию о стандартной библиотеке ввода-вывода. Эта строка встречается в начале многих исходных файлов Си-программ. Стандартная библиотека описана в главе 7 и приложении B.

Один из способов передачи данных между функциями состоит в том, что функция при обращении к другой функции передает ей список значений, называемых *аргументами*. Этот список берется в скобки и помещается после имени функции. В нашем примере *main* определена как функция, которая не ждет никаких аргументов, что отмечено пустым списком ().

Первая программа на Си:

`#include <stdio.h>`	Включение информации о стандартной библиотеке.
`main()`	Определение функции с именем *main*, не получающей никаких аргументов.
`{`	Инструкции *main* заключаются о фигурные скобки.
`printf("Hello, world\n");`	Функция *main* вызывает библиотечную функцию *printf* для печати заданной последовательности символов; \n – символ новой строки.
`}`	

Инструкции функции заключаются в фигурные скобки {}. Функция *main* содержит только одну инструкцию

```
printf("Hello, world\n");
```

Функция вызывается по имени, после которого, в скобках, указывается список аргументов. Таким образом, приведенная выше строка – это вызов функции *printf* с аргументом "Hello, world\n". Функция *printf* – это библиотечная функция, которая в данном случае напечатает последовательность символов, заключенную в двойные кавычки.

Последовательность символов в двойных кавычках, такая как "Hello, world\n", называется *строкой символов*, или *строковой константой*. Пока что в качестве аргументов для *printf* и других функций мы будем использовать только строки символов.

В Си комбинация **\n** внутри строки символов обозначает символ *новой строки* и при печати вызывает переход к левому краю следующей строки. Если вы удалите \n (стоит поэкспериментировать), то обнаружите, что, закончив печать, машина не переходит на новую строку. Символ новой строки в текстовый аргумент printf следует включать явным образом. Если вы попробуете выполнить, например,

```
printf("Hello, world
");
```

компилятор выдаст сообщение об ошибке.

Символ новой строки никогда не вставляется автоматически, так что одну строку можно напечатать по шагам с помощью нескольких обращений к *printf*. Нашу первую программу можно написать и так:

```
#include <stdio.h>

main()
{
    printf("Hello, ");
    printf("world");
    printf('\n');
}
```

В результате ее выполнения будет напечатана та же строка, что и раньше.

Заметим, что **\n** обозначает только один символ. Такие особые комбинации символов, начинающиеся с обратной наклонной черты, как **\n**, и называемые эскейп-последовательностями, широко применяются для обозначения трудно представимых или невидимых символов. Среди прочих в Си имеются символы **\t**, **\b**, **\"**, ****, обозначающие соответственно табуляцию, возврат на один символ назад (“забой” последнего символа), двойную кавычку, саму наклонную черту. Полный список таких символов представлен в параграфе 2.3.

Упражнение 1.1. Выполните программу, печатающую “Hello, world”, в вашей системе. Поэкспериментируйте, удаляя некоторые части программы, и посмотрите, какие сообщения об ошибках вы получите.

Упражнение 1.2. Выясните, что произойдет, если в строковую константу аргумента *printf* вставить \c, где c – символ, не входящий в представленный выше список.

1.2 Переменные и арифметические выражения

Приведенная ниже программа выполняет вычисления по формуле °C = (5/9)(°F–32) и печатает таблицу соответствия температур по Фаренгейту температурам по Цельсию:

```
0   –17
20  –6
40  4
60  15
80  26
100 37
120 48
140 60
160 71
180 82
200 93
220 104
240 115
260 126
280 137
300 148
```

Как и предыдущая, эта программа состоит из определения одной-единственной функции *main*. Она длиннее программы, печатающей “здравствуй, мир”, но по сути не сложнее. На ней мы продемонстрируем несколько новых возможностей, включая комментарий, объявления, переменные, арифметические выражения, циклы и форматный вывод.

```
#include <stdio.h>
/* печать таблицы температур по Фаренгейту
и Цельсию для fahr = 0, 20, ..., 300 */

main()
{
    int fahr, celsius;
    int lower, upper, step;

    lower = 0; /* нижняя граница таблицы температур */
    upper = 300; /* верхняя граница */
    step = 20;  /* шаг */

    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr-32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

Две строки:

```
/* печать таблицы температур по Фаренгейту
   и Цельсию для fahr = 0, 20, ... 300  */
```

являются *комментарием*, который в данном случае кратко объясняет, что делает программа. Все символы, помещенные между /* и */, игнорируются компилятором, и этим можно свободно пользоваться, чтобы сделать программу более понятной. Комментарий можно располагать в любом месте, где могут стоять символы пробела, табуляции или символ новой строки.

В Си любая переменная должна быть объявлена раньше, чем она будет использована; обычно все переменные объявляются в начале функции перед первой исполняемой инструкцией. В *объявлении* описываются свойства переменных. Оно состоит из названия типа и списка переменных, например:

```
int fahr, celsius;
int lower, upper, step;
```

Тип *int* означает, что значения перечисленных переменных есть целые, в отличие от него тип *float* указывает на значения с плавающей точкой, т. е. на числа, которые могут иметь дробную часть. Диапазоны значений обоих типов зависят от используемой машины.

Числа типа *int* бывают как 16-разрядные (лежат в диапазоне от -32768 до 32767), так и 32-разрядные. Числа типа *float* обычно представляются 32-разрядными словами, имеющими по крайней мере 6 десятичных значащих цифр (лежат приблизительно в диапазоне от 10^{-38} до 10^{+38}.

Помимо **int** и **float** в Си имеется еще несколько базовых типов для данных, это:

char - символ-единичный байт;
short - короткое целое;
long - длинное целое;
double - с плавающей точкой с двойной точностью.

Размеры объектов указанных типов также зависят от машины. Из базовых типов можно создавать: *массивы*, *структуры* и *объединения*, *указатели* на объекты базовых типов и функции, возвращающие значения этих типов в качестве результата. Обо всем этом мы расскажем позже.

Вычисления в программе преобразования температур начинаются с *инструкций присваивания*.

```
lower = 0;
upper = 300;
step  = 20;
fahr  = lower;
```

которые устанавливают указанные в них переменные в начальные значения. Любая инструкция заканчивается точкой с запятой.

Все строки таблицы вычисляются одним и тем же способом, поэтому мы воспользуемся циклом, повторяющим это вычисление для каждой строки. Необходимые действия выполнит цикл **while**:

```
while(fahr <= upper) {
    ...
}
```

Он работает следующим образом. Проверяется условие в скобках. Если оно истинно (значение *fahr* меньше или равно значению *upper*), то выполняется тело цикла (три инструкции, заключенные в фигурные скобки). Затем опять проверяется условие, и если оно истинно, то тело цикла выполняется снова. Когда условие становится ложным (*fahr* превысило *upper*), цикл завершается, и вычисления продолжаются с инструкции, следующей за циклом. Поскольку никаких инструкций за циклом нет, программа завершает работу.

Телом цикла *while* может быть одна или несколько инструкций, заключенных в фигурные скобки, как в программе преобразования температур, или одна-единственная инструкция без скобок, как в цикле

```
(while i < j)
    i = 2 * i;
```

И в том и в другом случае инструкции, находящиеся под управлением *while*, мы будем записывать со сдвигом, равным одной позиции табуляции, которая в программе указывается четырьмя пробелами; благодаря этому будут ясно видны инструкции, расположенные внутри цикла. Отступы подчеркивают логическую структуру программы. Си-компилятор не обращает внимания на внешнее оформление программы, но наличие в нужных местах отступов и пробелов существенно влияет на то, насколько легко она будет восприниматься человеком при просмотре. Чтобы лучше была видна логическая структура выражения, мы рекомендуем на каждой строке писать только по одной инструкции и с обеих сторон от операторов ставить пробелы. Положение скобок не так важно, хотя существуют различные точки зрения на этот счет. Мы остановились на одном из нескольких распространенных стилей их применения. Выберите тот, который больше всего вам нравится, и строго ему следуйте.

Большая часть вычислений выполняется в теле цикла. Температура по Фаренгейту переводится в температуру по Цельсию и присваивается переменной *celsius* посредством инструкции

```
celsius = 5 * (fahr-32) / 9;
```

Причина, по которой мы сначала умножаем на 5 и затем делим на 9, а не сразу умножаем на 5/9, связана с тем, что в Си, как и во многих других языках, деление целых сопровождается *отбрасыванием*, т. е. потерей дробной части. Так как 5 и 9 – целые, отбрасывание в 5/9 дало бы нуль, и на месте температур по Цельсию были бы напечатаны нули.

Этот пример прибавил нам еще немного знаний о том, как работает функция *printf*. Функция *printf* – это универсальная функция форматного ввода-вывода, которая будет подробно описана в главе 7. Ее первый аргумент – строка символов, в которой каждый символ % соответствует одному из последующих аргументов (второму, третьему, ...), а информация, расположенная за символом %, указывает на вид, в котором выводится каждый из этих аргументов. Например, %d специфицирует выдачу аргумента в виде целого десятичного числа, и инструкция

```
printf(“%d\t%d\n”, fahr, celsius);
```

печатает целое *fahr*, выполняет табуляцию (\t) и печатает целое *celsius*.

В функции *printf* каждому спецификатору первого аргумента (конструкции, начинающейся с **%**) соответствует второй аргумент, третий аргумент и т. д. Спецификаторы и соответствующие им аргументы должны быть согласованы по количеству и типам: в противном случае напечатано будет не то, что нужно.

Кстати, *printf* не является частью языка Си, и вообще в языке нет никаких специальных конструкций, определяющих ввод-вывод. Функция *printf* – лишь полезная функция стандартной библиотеки, которая обычно доступна для Си-программ. Поведение функции *printf*, однако, оговорено стандартом ANSI, и ее свойства должны быть одинаковыми во всех Си-системах, удовлетворяющих требованиям стандарта.

Желая сконцентрировать ваше внимание на самом Си, мы не будем много говорить о вводе-выводе до главы 7. В частности, мы отложим разговор о форматном вводе. Если вам потребуется ввести числа, советуем прочитать в параграфе 7.4 то, что касается функции *scanf*. Эта функция отличается от *printf* лишь тем, что она вводит данные, а не выводит.

Существуют еще две проблемы, связанные с программой преобразования температур. Одна из них (более простая) состоит в том, что выводимый результат выглядит несколько неряшливо, поскольку числа не выровнены по правой позиции колонок. Это легко исправить, добавив в каждый из спецификаторов формата %d указание о ширине поля; при этом программа будет печатать числа, прижимая их к правому краю указанных полей. Например, мы можем написать

```
printf(“%3d%6d\n”, fahr, celsius);
```

чтобы в каждой строке первое число печатать в поле из трех позиций, а второе – из шести. В результате будет напечатано:

```
  0   -17
 20    -6
 40     4
 60    15
 80    26
100    37
```

Вторая, более серьезная проблема связана с тем, что мы пользуемся целочисленной арифметикой и поэтому не совсем точно вычисляем температуры по шкале Цельсия. Например, 0°F на самом деле (с точностью до десятой) равно -17.8°C, а не -17. Чтобы получить более точные значения температур, нам надо пользоваться не целочисленной арифметикой, а арифметикой с плавающей точкой. Это потребует некоторых изменений в программе.

```
#include <stdio.h>
/* печать таблицы температур по Фаренгейту и Цельсию для
   fahr = 0, 20, . .., 300; вариант с плавающей точкой   */
main()
{
    float fahr, celsius;
    int lower, upper, step;

    lower = 0;   /* нижняя граница таблицы температур */
    upper = 300; /* верхняя граница */
    step = 20;   /* шаг */

    fahr = lower;
    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%3.0f %6.1f\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

Программа мало изменилась. Она отличается от предыдущей лишь тем, что *fahr* и *celsius* объявлены как **float**, а формула преобразования написана в более естественном виде. В предыдущем варианте нельзя было писать 5/9, так как целочисленное деление в результате обрезания дало бы нуль. Десятичная точка в константе указывает на то, что последняя рассматривается как число с плавающей точкой, и 5.0/9.0, таким образом, есть частное от деления двух значений с плавающей точкой, которое не предполагает отбрасывания дробной части. В том случае, когда арифметическая операция имеет целые операнды, она выполняется по правилам целочисленной арифметики. Если же один операнд с плавающей точкой, а другой - целый, то перед тем, как операция будет выполнена, последний будет преобразован в число с плавающей точкой. Если бы мы написали *fahr-32* то 32 автоматически было бы преобразовано в число с плавающей точкой. Тем не менее при записи констант с плавающей точкой мы всегда используем десятичную точку, причем даже в тех случаях, когда константы на самом деле имеют целые значения. Это делается для того, чтобы обратить внимание читающего программу на их природу.

Более подробно правила, определяющие, в каких случаях целые переводятся в числа с плавающей точкой, рассматриваются в главе 2. А сейчас заметим, что присваивание

```
fahr=lower;
```

и проверка

```
while(fahr <= upper)
```

работают естественным образом, т. е. перед выполнением операции значение *int* приводится к *float*.

Спецификация *%3.0f* в *printf* определяет печать числа с плавающей точкой (в данном случае числа *fahr*) в поле шириной не более трех позиций без десятичной точки и дробной части. Спецификация *%6.1f* описывает печать другого числа (*celsius*) в поле из шести позиций с одной цифрой после десятичной точки. Напечатано будет следующее:

```
  0  -17.8
 20   -6.7
 40    4.4
```

Ширину и точность можно не задавать; *%6f* означает, что число будет занимать не более шести позиций; *%.2f* - число имеет две цифры после десятичной точки, но ширина не ограничена; *%f* просто указывает на печать числа с плавающей точкой.

%d - печать десятичного целого.
%6d - печать десятичного целого в поле из шести позиций.
%f - печать числа с плавающей точкой.

%6f – печать числа с плавающей точкой в поле из шести позиций.
%.2f – печать числа с плавающей точкой с двумя цифрами после десятичной точки.
%6.2f– печать числа с плавающей точкой и двумя цифрами после десятичной точки в поле из шести позиций.

Кроме того, *printf* допускает следующие спецификаторы: **%o** для восьмеричного числа; **%x** для шестнадцатеричного числа; **%c** для символа; **%s** для строки символов и **%%** для самого %.

Упражнение 1.3. Усовершенствуйте программу преобразования температур таким образом, чтобы над таблицей она печатала заголовок.

Упражнение 1.4. Напишите программу, которая будет печатать таблицу соответствия температур по Цельсию температурам по Фаренгейту.

1.3 Инструкция for

Существует много разных способов для написания одной и той же программы. Видоизменим нашу программу преобразования температур:

```
#include <stdio.h>
/* печать таблицы температур по Фаренгейту и Цельсию */
main()
{
    int fahr;
    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Эта программа печатает тот же результат, но выглядит она, несомненно, по-другому. Главное отличие заключается в отсутствии большинства переменных. Осталась только переменная *fahr*, которую мы объявили как *int*. Нижняя и верхняя границы и шаг присутствуют в виде констант в инструкции *for* – новой для нас конструкции, а выражение, вычисляющее температуру по Цельсию, теперь задано третьим аргументом функции *printf*, а не в отдельной инструкции присваивания.

Последнее изменение является примером применения общего правила: в любом контексте, где возможно использовать значение переменной какого-то типа, можно использовать более сложное выражение того же типа. Так, на месте третьего аргумента функции *printf* согласно спецификатору *%6.1f* должно быть значение с плавающей точкой, следовательно, здесь может быть любое выражение этого типа.

Инструкция *for* описывает цикл, который является обобщением цикла *while*. Если вы сравните его с ранее написанным *while*, то вам станет ясно, как он работает. Внутри скобок имеются три выражения, разделяемые точкой с запятой. Первое выражение – инициализация

```
fahr = 0
```

выполняется один раз перед тем, как войти в цикл. Второе – проверка условия продолжения цикла

```
fahr <= 300
```

Условие вычисляется, и если оно истинно, выполняется тело цикла (в нашем случае это одно обращение к *printf*). Затем осуществляется приращение шага:

```
fahr = fahr + 20
```

и условие вычисляется снова. Цикл заканчивается, когда условие становится ложным. Как и в случае с *while*, тело *for*-цикла может состоять из одной инструкции или из нескольких, заключенных в фигурные скобки. На месте этих трех выражений (инициализации, условия и приращения шага) могут стоять произвольные выражения.

Выбор между *while* и *for* определяется соображениями ясности программы. Цикл *for* более удобен в тех случаях, когда инициализация и приращение шага логически связаны друг с другом общей переменной и выражаются единичными инструкциями, поскольку названный цикл компактнее цикла *while*, а его управляющие части сосредоточены в одном месте.

Упражнение 1.5. Измените программу преобразования температур так, чтобы она печатала таблицу в обратном порядке, т. е. от 300 до 0.

1.4 Именованные константы

Прежде чем мы закончим рассмотрение программы преобразования температур, выскажем еще одно соображение. Очень плохо, когда по программе рассеяны “загадочные числа”, такие как 300, 20. Тот, кто будет читать программу, не найдет в них и намека на то, что они собой представляют. Кроме того, их трудно заменить на другие каким-то систематическим способом. Одна из возможностей справиться с такими числами – дать им осмысленные имена. Строка **#define** определяет *символьное имя*, или *именованную константу*, для заданной строки символов:

```
#define имя подставляемый-текст
```

С этого момента при любом появлении *имени* (если только оно встречается не в тексте, заключенном в кавычки, и не является частью определения другого имени) оно будет заменяться на соответствующий ему *подставляемый-текст*. *Имя* имеет тот же вид, что и переменная: последовательность букв и цифр, начинающаяся с буквы. *Подставляемый-текст* может быть любой последовательностью символов, среди которых могут встречаться не только цифры.

```
#include <stdio.h>

#define LOWER 0   /* нижняя граница таблицы */
#define UPPER 300 /* верхняя граница */
#define STEP 20   /* размер шага */

/* печать таблицы температур по Фаренгейту и Цельсию */
main()
{
    int fahr;
    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf(“%3d %6.1f\n”, fahr, (5.0/9.0)*(fahr-32));
}
```

Величины *LOWER*, *UPPER* и *STEP* – именованные константы, а не переменные, поэтому для них нет объявлений. По общепринятому соглашению имена именованных констант набираются заглавными буквами, чтобы они отличались от обычных переменных, набираемых строчными. Заметим, что в конце *#define*-строки точка с запятой не ставится.

1.5 Ввод-вывод символов

Теперь мы намерены рассмотреть семейство программ по обработке текстов. Вы обнаружите, что многие существующие программы являются просто расширенными версиями обсуждаемых здесь прототипов.

Стандартная библиотека поддерживает очень простую модель ввода-вывода. Текстовый ввод-вывод вне зависимости от того, откуда он исходит или куда направляется, имеет дело с потоком символов. *Текстовый поток* – это последовательность символов, разбитая на строки, каждая из которых содержит нуль или более символов и завершается символом новой строки. Обязанность следить за тем, чтобы любой поток ввода-вывода отвечал этой модели, возложена на библиотеку: программист, пользуясь библиотекой, не должен заботиться о том, в каком виде строки представляются вне программы.

Стандартная библиотека включает несколько функций для чтения и записи одного символа. Простейшие из них – *getchar* и *putchar*. За одно обращение к *getchar* считывается *следующий символ ввода* из текстового потока, и этот символ выдается в качестве результата. Так, после выполнения

```
c = getchar();
```

переменная *c* содержит очередной символ ввода. Обычно символы поступают с клавиатуры. Ввод из файлов рассматривается в главе 7.

Обращение к *putchar* приводит к печати одного символа. Так,

```
putchar(c);
```

напечатает содержимое целой переменной *c* в виде символа (обычно на экране). Вызовы *putchar* и *printf* могут произвольным образом перемежаться. Вывод будет формироваться в том же порядке, что и обращения к этим функциям.

1.5.1 Копирование файла

При наличии функций *getchar* и *putchar*, ничего больше не зная о вводе-выводе, можно написать удивительно много полезных программ. Простейший пример – это программа, копирующая по одному символу с входного потока в выходной поток:

```
    чтение символа
    while (символ не является признаком конца файла)
    вывод только что прочитанного символа
    чтение символа
```

Оформляя ее в виде программы ни Си, получим

```
#include <stdio.h>

/* копирование ввода на вывод, 1-я версия */
main()
{
    int c;
    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

Оператор отношения **!=** означает “не равно”.

Каждый символ, вводимый с клавиатуры или появляющийся на экране, как и любой другой символ внутри машины, кодируется комбинацией битов. Тип **char** специально предназначен для хранения символьных данных, однако для этого также годится и любой целый тип. Мы пользуемся типом **int** и делаем это по одной важной причине, которая требует разъяснений.

Существует проблема: как отличить конец ввода от обычных читаемых данных. Решение заключается в том, чтобы функция *getchar* по исчерпании входного потока выдавала в качестве результата такое значение, которое нельзя было бы спутать ни с одним реальным символом. Это значение есть **EOF** (аббревиатура от *end of file* – конец файла). Мы должны объявить переменную *c* такого типа, чтобы его “хватило” для представления всех возможных результатов, выдаваемых функцией *getchar*. Нам не подходит тип *char*, так как *c* должна быть достаточно “емкой”, чтобы помимо любого значения типа *char* быть в состоянии хранить и *EOF*. Вот почему мы используем *int*, а не *char*.

EOF – целая константа, определенная в **<stdio.h>**. Какое значение имеет эта константа – неважно, лишь бы оно отличалось от любого из возможных значений типа *char*. Использование именованной константы с унифицированным именем гарантирует, что программа не будет зависеть от конкретного числового значения, которое, возможно, в других Си-системах будет иным.

Программу копирования можно написать более сжато. В Си любое присваивание, например

```
c = getchar()
```

трактуется как выражение со значением, равным значению левой части после присваивания. Это значит, что присваивание может встречаться внутри более сложного выражения. Если присваивание переменной *c* расположить в проверке условия цикла *while*, то программу копирования можно будет записать в следующем виде:

```
#include <stdio.h>
/* копирование ввода на вывод; 2-я версия */
main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

Цикл *while*, пересылая в *c* полученное от *getchar* значение, сразу же проверяет: не является ли оно “концом файла”. Если это не так, выполняется тело цикла *while* и печатается символ. По окончании ввода завершается работа цикла *while*, а тем самым и *main*.

В данной версии ввод “централизован”. – в программе имеется только одно обращение к *getchar*. В результате она более компактна и легче воспринимается при чтении. Вам часто придется сталкиваться с такой формой записи, где присваивание делается вместе с проверкой. (Чрезмерное увлечение ею, однако, может запутать программу, поэтому мы постараемся пользоваться указанной формой разумно.)

Скобки внутри условия, вокруг присваивания, необходимы. *Приоритет !=* выше, чем приоритет =, из чего следует, что при отсутствии скобок проверка != будет выполняться до операции присваивания =. Таким образом, запись

```
c = getchar() != EOF
```

эквивалентна записи

```
c = (getchar() != EOF)
```

А это совсем не то, что нам нужно: переменной *c* будет присваиваться 0 или 1 в зависимости от того, встретит или не встретит *getchar* признак конца файла. (Более подробно об этом см. в главе 2.)

Упражнение 1.6. Убедитесь в том, что выражение *getchar() != EOF* получает значение 0 или 1.

Упражнение 1.7. Напишите программу, печатающую значение *EOF*.

1.5.2 Подсчет символов

Следующая программа занимается подсчетом символов; она имеет много сходных черт с программой копирования.

```
#include <stdio.h>

/* подсчет вводимых символов; 1-я версия */
main()
{
    long nc;
    nc = 0;

    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

Инструкция

```
++nc;
```

представляет новый оператор **++**, который означает *увеличить на единицу*. Вместо этого можно было бы написать *nc=nc+1*, но *++nc* намного короче, а часто и эффективнее. Существует аналогичный оператор **--**, означающий *уменьшить на единицу*. Операторы **++** и **--** могут быть как префиксными (*++nc*), так и постфиксными (*nc++*). Как будет показано в главе 2, эти две формы в выражениях имеют разные значения, но и *++nc*, и *nc++* добавляют к *nc* единицу. В данном случае мы остановились на префиксной записи.

Программа подсчета символов накапливает сумму в переменной типа *long*. Целые типа *long* имеют не менее 32 битов. Хотя на некоторых машинах типы *int* и *long* имеют одинаковый размер, существуют, однако, машины, в которых *int* занимает 16 бит с максимально возможным значением 32767, а это – сравнительно маленькое число, и счетчик типа *int* может переполниться. Спецификация *%ld* в *printf* указывает, что соответствующий аргумент имеет тип *long*.

Возможно охватить еще больший диапазон значений, если использовать тип *double* (т. е. *float* с двойной точностью). Применим также инструкцию *for* вместо *while*, чтобы продемонстрировать другой способ написания цикла.

```
#include <stdio.h>

/* подсчет вводимых символов; 2-й версия */
main()
{
    double nc;
    for (nc = 0; getchar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

В *printf* спецификатор *%f* применяется как для *float*, так и для *double*; спецификатор *%.0f* означает печать без десятичной точки и дробной части (последняя в нашем случае отсутствует).

Тело указанного *for*-цикла пусто, поскольку кроме проверок и приращений счетчика делать ничего не нужно. Но правила грамматики Си требуют, чтобы *for*- цикл имел тело. Выполнение этого требования обеспечивает изолированная точка с запятой, называемая *пустой инструкцией*. Мы поставили точку с запятой на отдельной строке для большей наглядности.

Наконец, заметим, что если ввод не содержит ни одного символа, то при первом же обращении к *getchar* условие в *while* или *for* не будет выполнено и программа выдаст нуль, что и будет правильным результатом. Это важно. Одно из привлекательных свойств циклов *while* и *for* состоит в том, что условие проверяется до того, как выполняется тело цикла. Если ничего делать не надо, то ничего делаться и не будет, пусть даже

тело цикла не выполнится ни разу. Программа должна вести себя корректно и при нулевом количестве вводимых символов. Само устройство циклов *while* и *for* дает дополнительную уверенность в правильном поведении программы в случае граничных условий.

1.5.3 Подсчет строк

Следующая программа подсчитывает строки. Как упоминалось выше, стандартная библиотека обеспечивает такую модель ввода-вывода, при которой входной текстовый поток состоит из последовательности строк, каждая из которых заканчивается символом новой строки. Следовательно, подсчет строк сводится к подсчету числа символов новой строки.

```
#include <stdio.h>
/* подсчет строк входного потока */
main()
{
    int c, nl;
    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

Тело цикла теперь образует инструкция *if*, под контролем которой находится увеличение счетчика *nl* на единицу. Инструкция if проверяет условие в скобках и, если оно истинно, выполняет следующую за ним инструкцию (или группу инструкций, заключенную в фигурные скобки). Мы опять делаем отступы в тексте программы, чтобы показать, что чем управляется.

Двойной знак равенства в языке Си обозначает оператор "равно" (он аналогичен оператору = в Паскале и *.EQ.* в Фортране). Удваивание знака = в операторе проверки на равенство сделано для того, чтобы отличить его от единичного =, используемого в Си для обозначения присваивания. Предупреждаем: начинающие программировать на Си иногда пишут =, а имеют в виду ==. Как мы увидим в главе 2, в этом случае результатом будет обычно вполне допустимое по форме выражение, на которое компилятор не выдаст никаких предупреждающих сообщений (Современные компиляторы, как правило, выдают предупреждение о возможной ошибке. – *Примеч. ред.*).

Символ, заключенный в одиночные кавычки, представляет собой целое значение, равное коду этого символа (в кодировке, принятой на данной машине). Это так называемая *символьная константа*. Существует и другой способ для написания маленьких целых значений. Например, 'A' есть символьная константа, в наборе символов ASCII ее значение равняется 65 – внутреннему представлению символа *A*. Конечно, 'A' в роли константы предпочтительнее, чем 65, поскольку смысл первой записи более очевиден, и она не зависит от конкретного способа кодировки символов.

Эскейп-последовательности, используемые в строковых константах, допускаются также и в символьных константах. Так, **'\n'** обозначает код символа новой строки, который в ASCII равен 10. Следует обратить особое внимание на то, что **'\n'** обозначает один символ (код которого в выражении рассматривается как целое значение), в то время как **"\n"** – строковая константа, в которой чисто случайно указан один символ. Более подробно различие между символьными и строковыми константами разбирается в главе 2.

Упражнение 1.8. Напишите программу для подсчета пробелов, табуляций и новых строк.

Упражнение 1.9. Напишите программу, копирующую символы ввода в выходной поток и заменяющую стоящие подряд пробелы на один пробел.

Упражнение 1.10. Напишите программу, копирующую вводимые символы в выходной поток с заменой символа табуляции на **\t**, символа забоя на **\b** и каждой обратной наклонной черты на ****. Это сделает видимыми все символы табуляции и забоя.

1.5.4 Подсчет слов

Четвертая из нашей серии полезных программ подсчитывает строки, слова и символы, причем под словом здесь имеется в виду любая строка символов, не содержащая в себе пробелов, табуляций и символов новой строки. Эта программа является упрощенной версией программы **wc** системы UNIX.

```
#include <stdio.h>

#define IN 1    /* внутри слова */
#define OUT 0   /* вне слова */
```

```
/* подсчет строк, слов и символов */
main()
{
    int c, nl, nw, nc, state;
    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

Каждый раз, встречая первый символ слова, программа изменяет значение счетчика слов на 1. Переменная *state* фиксирует текущее состояние – находимся мы внутри или вне слова. Вначале ей присваивается значение OUT, что соответствует состоянию "вне слова". Мы предпочитаем пользоваться именованными константами IN и OUT, а не собственно значениями 1 и 0, чтобы сделать программу более понятной. В такой маленькой программе этот прием мало что дает, но в большой программе увеличение ее ясности окупает незначительные дополнительные усилия, потраченные на то, чтобы писать программу в таком стиле с самого начала. Вы обнаружите, что большие изменения гораздо легче вносить в те программы, в которых магические числа встречаются только в виде именованных констант.

Строка

```
nl = nw = nc = 0;
```

устанавливает все три переменные в нуль. Такая запись не является какой-то особой конструкцией и допустима потому, что присваивание есть выражение со своим собственным значением, а операции присваивания выполняются справа налево. Указанная строка эквивалентна

```
nl = (nw = (nc = 0));
```

Оператор `||` означает **ИЛИ**, так что строка

```
if (c == ' ' || c == '\n' || c == '\t' )
```

читается как "если с есть пробел, *или* с есть новая строка, *или* с есть табуляция". (Напомним, что видимая эскейп-последовательность \t обозначает символ табуляции.) Существует также оператор **&&**, означающий **И**. Его приоритет выше, чем приоритет `||`. Выражения, связанные операторами && или ||, вычисляются слева направо; при этом гарантируется, что вычисления сразу прервутся, как только будет установлена истинность или ложность условия. Если с есть пробел, то дальше проверять, является значение с символом новой строки или же табуляции, не нужно. В этом частном случае данный способ вычислений не столь важен, но он имеет значение в более сложных ситуациях, которые мы вскоре рассмотрим.

В примере также встречается слово **else**, которое указывает на альтернативные действия, выполняемые в случае, когда условие, указанное в **if**, не является истинным. В общем виде условная инструкция записывается так:

if (*выражение*)
    инструкция$_1$
else
    инструкция$_2$

В конструкции **if-else** выполняется одна и только одна из двух инструкций. Если *выражение* истинно, то выполняется *инструкция*$_1$, если нет, то – *инструкция*$_2$. Каждая из этих двух инструкций представляет собой либо одну инструкцию, либо несколько, заключенных в фигурные скобки. В нашей программе после *else* стоит инструкция *if*, управляющая двумя такими инструкциями.

Упражнение 1.11. Как протестировать программу подсчета слов? Какой ввод вероятнее всего обнаружит ошибки, если они были допущены?

Упражнение 1.12. Напишите программу, которая печатает содержимое своего ввода, помещая по одному слову на каждой строке.

1.6 Массивы

А теперь напишем программу, подсчитывающую по отдельности каждую цифру, символы-разделители (*пробелы, табуляции и новые-строки*) и все другие символы. Это несколько искусственная программа, но она позволит нам в одном примере продемонстрировать еще несколько возможностей языка Си. Имеется двенадцать категорий вводимых символов. Удобно все десять счетчиков цифр хранить в массиве, а не в виде десяти отдельных переменных. Вот один из вариантов этой программы:

```
#include <stdio.h>

/* подсчет цифр, символов-разделителей и прочих символов */
main()
{
    int c, i, nwhite, nother;
    int ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10, ++i)
        ndigit[i]= 0;

    while ((c = getchar()) != EOF)
        if (c >='0' && c <= '9')
            ++ndigit[c – '0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;

    printf(“цифры =“);
    for (i = 0; i < 10; ++i)
        printf(“%d”, ndigit[i]);
    printf(“, символы-разделители =%d, прочие =%d\n”, nwhite, nother);
}
```

В результате выполнения этой программы будет напечатан следующий результат:

```
цифры = 9 3 0 0 0 0 0 0 0 1, символы-разделители = 123, прочие = 345
```

Объявление

```
int ndigit[10];
```

объявляет *ndigit* массивом из 10 значений типа *int*. В Си элементы массива всегда нумеруются начиная с нуля, так что элементами этого массива будут ndigit[0], ndigit[1], ..., ndigit[9], что учитывается в *for*-циклах (при инициализации и печати массива).

Индексом может быть любое целое выражение, образуемое целыми переменными (например *i*) и целыми константами.

Приведенная программа опирается на определенные свойства кодировки цифр. Например, проверка

```
if (c >= '0' && c <= '9') ...
```

определяет, является ли находящийся в *c* символ цифрой. Если это так, то

```
c –'0'
```

есть числовое значение цифры. Сказанное справедливо только в том случае, если для ряда значений '0','1', ..., '9' каждое следующее значение на 1 больше предыдущего. К счастью, это правило соблюдается во всех наборах символов.

По определению, значения типа *char* являются просто малыми целыми, так что переменные и константы типа *char* в арифметических выражениях идентичны значениям типа *int*. Это и естественно, и удобно; например, c–'0' есть целое выражение с возможными значениями от 0 до 9, которые соответствуют символам от '0' до '9', хранящимся в переменной c. Таким образом, значение данного выражения является правильным индексом для массива ndigit.

Следующий фрагмент определяет, является символ цифрой, символом-разделителем или чем-нибудь иным.

```
if (c >= '0' && c <= '9')
    ++n[c-'0'];
else if (c ==' ' || c == '\n' || c == '\t')
    ++nwhite;
else
    ++nother;
```

Конструкция вида

```
if (условие1)
    инструкция1
else if (условие2)
    инструкция2
…
  …
else
    инструкцияn
```

часто применяется для выбора одного из нескольких альтернативных путей, имеющихся в программе. *Условия* вычисляются по порядку в направлении сверху вниз до тех пор, пока одно из них не будет удовлетворено; в этом случае будет выполнена соответствующая ему *инструкция*, и работа всей конструкции завершится. (Любая из инструкций может быть группой инструкций в фигурных скобках.) Если ни одно из условий не удовлетворено, выполняется последняя инструкция, расположенная сразу за *else*, если таковая имеется. Если же *else* и следующей за ней инструкции нет (как это было в программе подсчета слов), то никакие действия вообще не производятся. Между первым *if* и завершающим *else* может быть сколько угодно комбинаций вида

```
else if (условие)
    инструкция
```

Когда их несколько, программу разумно форматировать так, как мы здесь показали. Если же каждый следующий *if* сдвигать вправо относительно предыдущего *else*, то при длинном каскаде проверок текст окажется слишком близко прижатым к правому краю страницы.

Инструкция **switch**, речь о которой пойдет в главе 3, обеспечивает другой способ изображения многопутевого ветвления на языке Си. Он более подходит, в частности, тогда, когда условием перехода служит совпадение значения некоторого выражения целочисленного типа с одной из констант, входящих в заданный набор. Вариант нашей программы, реализованной с помощью **switch**, приводится в параграфе 3.4.

Упражнение 1.13. Напишите программу, печатающую гистограммы длин вводимых слов. Гистограмму легко рисовать горизонтальными полосами. Рисование вертикальными полосами – более трудная задача.

Упражнение 1.14. Напишите программу, печатающую гистограммы частот встречаемости вводимых символов.

1.7 Функции

Функции в Си играют ту же роль, что и подпрограммы и функции в Фортране или процедуры и функции в Паскале. Функция обеспечивает удобный способ отдельно оформить некоторое вычисление и пользоваться им далее, не заботясь о том, как оно реализовано. После того, как функции написаны, можно забыть, *как* они сделаны, достаточно знать лишь, *что* они умеют делать. Механизм использования функции в Си удобен, легок и эффективен. Нередко вы будете встречать короткие функции, вызываемые лишь единожды: они оформлены в виде функции с одной-единственной целью – получить более ясную программу.

До сих пор мы пользовались готовыми функциями вроде main, getchar и putchar, теперь настала пора нам самим написать несколько функций. В Си нет оператора возведения в степень вроде ** в Фортране. Поэтому проиллюстрируем механизм определения функции на примере функции **power(m, n)**, которая возводит целое m в целую положительную степень n. Так, power(2, 5) имеет значение 32. На самом деле для практического применения эта функция малопригодна, так как оперирует лишь малыми целыми степенями, однако она вполне может послужить иллюстрацией. (В стандартной библиотеке есть функция **pow(x, y)**, вычисляющая x в степени y.)

Итак, мы имеем функцию power и главную функцию main, пользующуюся ее услугами, так что вся программа выглядит следующим образом:

```
#include <stdio.h>

int power(int m, int n);

/* тест функции power */
main()
{
    int i;
    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2,i), power(-3,i));
    return 0;
}
```

```
/* возводит base в n-ю степень, n >= 0 */
int power(int base, int n)
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Определение любой функции имеет следующий вид:

```
тип-результата имя-функции (список параметров, если он есть)
{
    объявления
    инструкции
}
```

Определения функций могут располагаться в любом порядке в одном или в нескольких исходных файлах, но любая функция должна быть целиком расположена в каком-то одном. Если исходный текст программы распределен по нескольким файлам, то, чтобы ее скомпилировать и загрузить, вам придется сказать несколько больше, чем при использовании одного файла; но это уже относится к операционной системе, а не к языку. Пока мы предполагаем, что обе функции находятся в одном файле, так что будет достаточно тех знаний, которые вы уже получили относительно запуска программ на Си.

В следующей строке из функции main к power обращаются дважды.

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

При каждом вызове функции power передаются два аргумента, и каждый раз главная программа main в ответ получает целое число, которое затем приводится к должному формату и печатается. Внутри выражения power(2, i) представляет собой целое значение точно так же, как 2 или i. (Не все функции в качестве результата выдают целые значения; подробно об этом будет сказано в главе 4.)

В первой строке определения power:

```
int power(int base, int n);
```

указываются типы параметров, имя функции и тип результата. Имена параметров локальны внутри *power*, это значит, что они скрыты для любой другой функции, так что остальные подпрограммы могут свободно пользоваться теми же именами для своих целей. Последнее утверждение справедливо также для переменных i и p: i в power и i в main не имеют между собой ничего общего.

Далее *параметром* мы будем называть переменную из списка параметров, заключенного в круглые скобки и заданного в определении функции, а *аргументом* – значение, используемое при обращении к функции. Иногда в том же смысле мы будем употреблять термины *формальный аргумент* и *фактический аргумент*.

Значение, вычисляемое функцией power, возвращается в main с помощью инструкции **return**. За словом *return* может следовать любое выражение:

```
return выражение;
```

Функция не обязательно возвращает какое-нибудь значение. Инструкция *return* без выражения только передает управление в ту программу, которая ее вызвала, не передавая ей никакого результирующего значения. То же самое происходит, если в процессе вычислений мы выходим на конец функции, обозначенный в тексте последней закрывающей фигурной скобкой. Возможна ситуация, когда вызывающая функция игнорирует возвращаемый ей результат.

Вы, вероятно, обратили внимание на инструкцию *return* в конце *main*. Поскольку main есть функция, как и любая другая она может вернуть результирующее значение тому, кто ее вызвал, – фактически в ту среду, из которой была запущена программа. Обычно возвращается нулевое значение, что говорит о нормальном завершении выполнения. Ненулевое значение сигнализирует о необычном или ошибочном завершении. До сих пор ради простоты мы опускали return в main, но с этого момента будем задавать return как напоминание о том, что программы должны сообщать о состоянии своего завершения в операционную систему.

Объявление

```
int power(int m, int n);
```

стоящее непосредственно перед *main*, сообщает, что функция power ожидает двух аргументов типа *int* и возвращает результат типа *int*. Это объявление, называемое *прототипом функции*, должно быть согласовано с определением и всеми вызовами *power*. Если определение функции или вызов не соответствует своему прототипу, это ошибка.

Имена параметров не требуют согласования. Фактически в прототипе они могут быть произвольными или вообще отсутствовать, т. е. прототип можно было бы записать и так:

```
int power(int, int);
```

Однако удачно подобранные имена поясняют программу, и мы будем часто этим пользоваться.

Историческая справка. Самые большие отличия ANSI-Си от более ранних версий языка как раз и заключаются в способах объявления и определения функций. В первой версии Си функцию power требовалось задавать в следующем виде:

```
/* power: возводит base в n-ю степень, n >= 0 */
/*          (версия в старом стиле языка Си) */
power(base, n)
int base, n;
{
    int i, p;
    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Здесь имена параметров перечислены в круглых скобках, а их типы заданы перед первой открывающей фигурной скобкой. В случае отсутствия указания о типе параметра, считается, что он имеет тип *int*. (Тело функции не претерпело изменений.)

Описание power в начале программы согласно первой версии Си должно было бы выглядеть следующим образом:

```
int power();
```

Нельзя было задавать список параметров, и поэтому компилятор не имел возможности проверить правильность обращений к power. Так как при отсутствии объявления power предполагалось, что функция возвращает значение типа int, то в данном случае объявление целиком можно было бы опустить.

Новый синтаксис для прототипов функций облегчает компилятору обнаружение ошибок в количестве аргументов и их типах. Старый синтаксис объявления и определения функции все еще допускается стандартом ANSI, по крайней мере на переходный период, но если ваш компилятор поддерживает новый синтаксис, мы настоятельно рекомендуем пользоваться только им.

Упражнение 1.15. Перепишите программу преобразования температур, выделив само преобразование в отдельную функцию.

1.8 Аргументы. Вызов по значению

Одно свойство функций в Си, вероятно, будет в новинку для программистов, которые уже пользовались другими языками, в частности Фортраном. В Си все аргументы функции передаются “по значению”. Это следует понимать так, что вызываемой функции посылаются значения ее аргументов во временных переменных, а не сами аргументы. Такой способ передачи аргументов несколько отличается от “вызова по ссылке” в Фортране и спецификации var при параметре в Паскале, которые позволяют подпрограмме иметь доступ к самим аргументам, а не к их локальным копиям.

Главное отличие заключается в том, что в Си вызываемая функция не может непосредственно изменить переменную вызывающей функции: она может изменить только ее частную, временную копию.

Однако вызов по значению следует отнести к достоинствам языка, а не к его недостаткам. Благодаря этому свойству обычно удается написать более компактную программу, содержащую меньшее число посторонних переменных, поскольку параметры можно рассматривать как должным образом инициализированные локальные переменные вызванной подпрограммы. В качестве примера приведем еще одну версию функции power, в которой как раз использовано это свойство.

```
/* power: возводит base в n-ю степень; n >= 0, версия 2 */
int power(int base, int n)
{
```

```
    int p;
    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

Параметр n выступает здесь в роли временной переменной, в которой циклом *for* в убывающем порядке ведется счет числа шагов до тех пор, пока ее значение не станет нулем. При этом отпадает надобность в дополнительной переменной i для счетчика цикла. Что бы мы ни делали с n внутри power, это не окажет никакого влияния на сам аргумент, копия которого была передана функции power при ее вызове.

При желании можно сделать так, чтобы функция смогла изменить переменную в вызывающей программе. Для этого последняя должна передать адрес подлежащей изменению переменной (*указатель* на переменную), а в вызываемой функции следует объявить соответствующий параметр как указатель и организовать через него косвенный доступ к этой переменной. Все, что касается указателей, мы рассмотрим в главе 5.

Механизм передачи массива в качестве аргумента несколько иной. Когда аргументом является имя массива, то функции передается значение, которое является адресом начала этого массива; никакие элементы массива не копируются. С помощью индексирования относительно полученного значения функция имеет доступ к любому элементу массива. Разговор об этом пойдет в следующем параграфе.

1.9 Символьные массивы

Самый распространенный вид массива в Си – массив символов. Чтобы проиллюстрировать использование символьных массивов и работающих с ними функций, напишем программу, которая читает набор текстовых строк и печатает самую длинную из них. Ее схема достаточно проста:

```
while (есть ли еще строка?)
    if (данная строка длиннее самой длинной из предыдущих)
        запомнить ее
        запомнить ее длину
напечатать самую длинную строку
```

Из схемы видно, что программа естественным образом распадается на части. Одна из них получает новую строку, другая проверяет ее, третья запоминает, а остальные управляют процессом вычислений.

Поскольку процесс четко распадается на части, хорошо бы так и перевести его на Си. Поэтому сначала напишем отдельную функцию getline для получения очередной строки. Мы попытаемся сделать эту функцию полезной и для других применений. Как минимум getline должна сигнализировать о возможном конце файла, а еще лучше, если она будет выдавать длину строки – или нуль в случае исчерпания файла. Нуль годится для признака конца файла, поскольку не бывает строк нулевой длины, даже строка, содержащая только один символ новой строки, имеет длину 1.

Когда мы обнаружили строку более длинную, чем самая длинная из всех предыдущих, то нам надо будет где-то ее запомнить. Здесь напрашивается вторая функция, copy, которая умеет копировать новую строку в надежное место.

Наконец, нам необходима главная программа, которая бы управляла функциями getline и copy. Вот как выглядит наша программа в целом:

```
#include <stdio.h>
#define MAXLINE 1000 /* максимальный размер вводимой строки */

int getline(char line[], int MAXLINE);
void copy(char to[], char from[]);

/* печать самой длинной строки */
main()
{
    int len; /* длина текущей строки */
    int max; /* длина максимальной из просмотренных строк */
    char line[MAXLINE]; /* текущая строка */
    char longest[MAXLINE]; /* самая длинная строка */

    max = 0;
    while (len = getline(line, MAXLINE)) > 0)
        if (len > max) {
            max = len;
            copy(longest, line);
        }
    if (max > 0) /* была ли хоть одна строка? */
        printf("%s", longest);
```

```
    return 0;
}


/* getline: читает строку в s, возвращает длину */
int getline(char s[], int lim)
{
    int c, i;

    for (i = 0; i < lim-1 && (c = getchar()) != EOF && c != '\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
}

/* copy: копирует из 'from' в 'to'; to достаточно большой */
void copy(char to[], char from[])
{
    int i;

    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```

Мы предполагаем, что функции *getline* и *copy*, описанные в начале программы, находятся в том же файле, что и *main*.

Функции *main* и *getline* взаимодействуют между собой через пару аргументов и возвращаемое значение. В *getline* аргументы определяются строкой

```
int getline(char s[], int lim);
```

Как мы видим, ее первый аргумент *s* есть массив, а второй, *lim*, имеет тип *int*. Задание размера массива в определении имеет целью резервирование памяти. В самой *getline* задавать длину массива *s* нет необходимости, так как его размер указан в *main*. Чтобы вернуть значение вызывающей программе, *getline* использует *return* точно так же, как это делает функция *power*. В приведенной строке также сообщается, что *getline* возвращает значение типа *int*, но так как при отсутствии указания о типе подразумевается *int*, то перед *getline* слово *int* можно опустить.

Одни функции возвращают результирующее значение, другие (такие как *copy*) нужны только для того, чтобы произвести какие-то действия, не выдавая никакого значения. На месте типа результата в *copy* стоит **void**. Это явное указание на то, что никакого значения данная функция не возвращает.

Функция *getline* в конец создаваемого ею массива помещает символ **'\0'** (**null**-символ, кодируемый нулевым байтом), чтобы пометить конец строки символов. То же соглашение относительно окончания нулем соблюдается и в случае строковой константы вроде

```
“hello\n”
```

В данном случае для него формируется массив из символов этой строки с '\0' в конце.

h e l l o \n \0

Спецификация *%s* в формате *printf* предполагает, что соответствующий ей аргумент – строка символов, оформленная указанным выше образом. Функция *copy* в своей работе также опирается на тот факт, что читаемый ею аргумент заканчивается символом '\0', который она копирует наряду с остальными символами. (Всё сказанное предполагает, что '\0' не встречается внутри обычного текста.)

Попутно стоит заметить, что при работе даже с такой маленькой программой мы сталкиваемся с некоторыми конструктивными трудностями. Например, что должна делать *main*, если встретится строка, превышающая допустимый размер? Функция *getline* работает надежно: если массив полон, она прекращает пересылку, даже если символа новой строки не обнаружила. Получив от *getline* длину строки и увидев, что она совпадает с MAXLINE, главная программа main могла бы “отловить” этот особый случай и справиться с ним. В интересах краткости описание этого случая мы здесь опускаем.

Пользователи *getline* не могут заранее узнать, сколь длинными будут вводимые строки, поэтому *getline* делает проверки на переполнение. А вот пользователям функции *copy* размеры копируемых строк известны (или они могут их узнать), поэтому дополнительный контроль здесь не нужен.

Упражнение 1.16. Перепишите main предыдущей программы так, чтобы она могла печатать самую длинную строку без каких-либо ограничений на ее размер.

Упражнение 1.17. Напишите программу печати всех вводимых строк, содержащих более 80 символов.

Упражнение 1.18. Напишите программу, которая будет в каждой вводимой строке заменять стоящие подряд символы пробелов и табуляций на один пробел и удалять пустые строки.

Упражнение 1.19. Напишите функцию *reverse(s)*, размещающую символы в строке *s* в обратном порядке. Примените ее при написании программы, которая каждую вводимую строку располагает в обратном порядке.

1.10 Внешние переменные и область видимости

Переменные *line*, *longest* и прочие принадлежат только функции *main*, или, как говорят, локальны в ней. Поскольку они объявлены внутри *main*, никакие другие функции прямо к ним обращаться не могут. То же верно и применительно к переменным других функций. Например, *i* в *getline* не имеет никакого отношения к *i* в *copy*. Каждая локальная переменная функции возникает только в момент обращения к этой функции и исчезает после выхода из нее. Вот почему такие переменные, следуя терминологии других языков, называют *автоматическими*. (В главе 4 обсуждается класс памяти **static**, который позволяет локальным переменным сохранять свои значения в промежутках между вызовами.)

Так как автоматические переменные образуются и исчезают одновременно с входом в функцию и выходом из нее, они не сохраняют своих значений от вызова к вызову и должны устанавливаться заново при каждом новом обращении к функции. Если этого не делать, они будут содержать “мусор”.

В качестве альтернативы автоматическим переменным можно определить *внешние* переменные, к которым разрешается обращаться по их именам из любой функции. (Этот механизм аналогичен области COMMON в Фортране и определениям переменных в самом внешнем блоке в Паскале.) Так как внешние переменные доступны повсеместно, их можно использовать вместо аргументов для связи между функциями по данным. Кроме того, поскольку внешние переменные существуют постоянно, а не возникают и исчезают на период выполнения функции, свои значения они сохраняют и после возврата из функций, их установивших.

Внешняя переменная должна быть *определена*, причем только один раз, вне текста любой функции; в этом случае ей будет выделена память. Она должна быть *объявлена* во всех функциях, которые хотят ею пользоваться. Объявление содержит сведения о типе переменной. Объявление может быть явным, в виде инструкции **extern**, или неявным, когда нужная информация получается из контекста. Чтобы конкретизировать сказанное, перепишем программу печати самой длинной строки с использованием *line*, *longest* и *max* в качестве внешних переменных. Это потребует изменений в вызовах, объявлениях и телах всех трех функций.

```
#include <stdio.h>

#define MAXLINE 1000 /* максимальный размер вводимой строки */

int max; /* длина максимальной из просмотренных строк */
char line[MAXLINE]; /* текущая строка */
char longest[MAXLINE]; /* самая длинная строка */

int getline(void);
void copy(void);

/* печать самой длинной строки: специализированная версия */
main()
{
    int len;
    extern int max;
    extern char longest[];
    max = 0;
    while ((len = getline()) > 0)
        if (len > max) {
            max = len;
            copy();
        }
    if (max > 0) /* была хотя бы одна строка */
        printf(“%s”, longest);
    return 0;
}

/* getline: специализированная версия */
int getline(void)
{
    int c, i;
    extern char line[];
    for (i = 0; i < MAXLINE-1
```

```
        && (c=getchar()) != EOF && c != '\n'; ++i)
            line[i] = c;
    if(c == '\n') {
        line[i]= c;
         ++i;
    }
    line[i] = '\0';
    return i;
}

/* copy: специализированная версия */
void copy(void)
{
    int i;
    extern char line[], longest[];

    i = 0;
    while ((longest[i] = line[i]) != '\0')
        ++i;
}
```

Внешние переменные для *main, getline* и *copy* определяются в начале нашего примера, где им присваивается тип и выделяется память. Определения внешних переменных синтаксически ничем не отличаются от определения локальных переменных, но поскольку они расположены вне функций, эти переменные считаются внешними. Чтобы функция могла пользоваться внешней переменной, ей нужно прежде всего сообщить имя соответствующей переменной. Это можно сделать, например, задав объявление *extern*, которое по виду отличается от объявления внешней переменной только тем, что оно начинается с ключевого слова *extern*.

В некоторых случаях объявление *extern* можно опустить. Если определение внешней переменной в исходном файле расположено выше функции, где она используется, то в объявлении *extern* нет необходимости. Таким образом, в *main, getline* и *copy* объявления *extern* избыточны. Обычно определения внешних переменных располагают в начале исходного файла, и все объявления *extern* для них опускают.

Если же программа расположена в нескольких исходных файлах и внешняя переменная определена в *файле1*, а используется в *файле2* и *файле3*, то объявления *extern* в *файле2* и *файле3* обязательны, поскольку необходимо указать, что во всех трех файлах функции обращаются к одной и той же внешней переменной. На практике обычно удобно собрать все объявления внешних переменных и функций в отдельный файл, называемый **заголовочным** (*header* -файлом), и помещать его с помощью **#include** в начало каждого исходного файла. В именах header-файлов по общей договоренности используется суффикс **.h**. В этих файлах, в частности в *<stdio.h>*, описываются также функции стандартной библиотеки. Более подробно о заголовочных файлах говорится в главе 4, а применительно к стандартной библиотеке – в главе 7 и приложении B.

Так как специализированные версии getline и copy не имеют аргументов, на первый взгляд кажется, что логично их прототипы задать в виде getline() и copy(). Но из соображений совместимости со старыми Си-программами стандарт рассматривает пустой список как сигнал к тому, чтобы выключить все проверки на соответствие аргументов. Поэтому, когда нужно сохранить контроль и явно указать отсутствие аргументов, следует пользоваться словом **void**. Мы вернемся к этой проблеме в главе 4.

Заметим, что по отношению к внешним переменным в этом параграфе мы очень аккуратно используем понятия *определение* и *объявление*. “Определение” располагается в месте, где переменная создается и ей отводится память; “объявление” помещается там, где фиксируется природа переменной, но никакой памяти для нее не отводится.

Следует отметить тенденцию все переменные делать внешними. Дело в том, что, как может показаться на первый взгляд, это приводит к упрощению связей – ведь списки аргументов становятся короче, а переменные доступны везде, где они нужны; однако они оказываются доступными и там, где не нужны. Так что чрезмерный упор на внешние переменные чреват большими опасностями – он приводит к созданию программ, в которых связи по данным не очевидны, поскольку переменные могут неожиданным и даже таинственным способом изменяться. Кроме того, такая программа с трудом поддается модификациям. Вторая версия программы поиска самой длинной строки хуже, чем первая, отчасти по этим причинам, а отчасти из-за нарушения общности двух полезных функций, вызванного тем, что в них вписаны имена конкретных переменных, с которыми они оперируют.

Итак, мы рассмотрели то, что можно было бы назвать ядром Си. Описанных “кирпичиков” достаточно, чтобы создавать полезные программы значительных размеров, и было бы чудесно, если бы вы, прервав чтение, посвятили этому какое-то время. В следующих упражнениях мы предлагаем вам создать несколько более сложные программы, чем рассмотренные выше.

Упражнение 1.20. Напишите программу detab, заменяющую символы табуляции во вводимом тексте нужным числом пробелов (до следующего “стопа” табуляции). Предполагается, что “стопы” табуляции расставлены на

фиксированном расстоянии друг от друга, скажем, через n позиций. Как лучше задавать n – в виде значения переменной или в виде именованной константы?

Упражнение 1.21. Напишите программу entab, заменяющую строки из пробелов минимальным числом табуляций и пробелов таким образом, чтобы вид напечатанного текста не изменился. Используйте те же “стопы” табуляции, что и в detab. В случае, когда для выхода на очередной “стоп” годится один пробел, что лучше – пробел или табуляция?

Упражнение 1.22. Напишите программу, печатающую символы входного потока так, чтобы строки текста не выходили правее n-й позиции. Это значит, что каждая строка, длина которой превышает n, должна печататься с переносом на следующие строки. Место переноса следует “ искать” после последнего символа, отличного от символа- разделителя, расположенного левее n-й позиции. Позаботьтесь о том, чтобы ваша программа вела себя разумно в случае очень длинных строк, а также когда до n-й позиции не встречается ни одного символа пробела или табуляции.

Упражнение 1.23. Напишите программу, убирающую все комментарии из любой Си- программы. Не забудьте должным образом обработать строки символов и строковые константы. Комментарии в Си не могут быть вложены друг в друга.

Упражнение 1.24. Напишите программу, проверяющую Си-программы на элементарные синтаксические ошибки вроде несбалансированности скобок всех видов. Не забудьте о кавычках (одиночных и двойных), эскейп-последовательностях (\...) и комментариях. (Это сложная программа, если писать ее для общего случая.)

[Назад] [Содержание] [Вперед]

[Главная] [Гостевая]