Глава 4. Функции и структура программы

Функции разбивают большие вычислительные задачи на более мелкие и позволяют воспользоваться тем, что уже сделано другими разработчиками, а не начинать создание программы каждый раз "с нуля". В выбранных должным образом функциях "упрятаны" несущественные для других частей программы детали их функционирования, что делает программу в целом более ясной и облегчает внесение в нее изменений.

Язык проектировался так, чтобы функции были эффективными и простыми в использовании. Обычно программы на Си состоят из большого числа небольших функций, а не из немногих больших. Программу можно располагать в одном или нескольких исходных файлах. Эти файлы можно компилировать отдельно, а загружать вместе, в том числе и с ранее откомпилированными библиотечными функциями. Процесс загрузки здесь не рассматривается, поскольку он различен в разных системах.

Объявление и определение функции - это та область, где стандартом ANSI в язык внесены самые существенные изменения. Как мы видели в главе 1, в описании функции теперь разрешено задавать типы аргументов. Синтаксис определения функции также изменен, так что теперь объявления и определения функций соответствуют друг другу. Это позволяет компилятору обнаруживать намного больше ошибок, чем раньше. Кроме того. если типы аргументов соответствующим образом объявлены, то необходимые преобразования аргументов выполняются автоматически.

Стандарт вносит ясность в правила, определяющие области видимости имен; в частности, он требует, чтобы для каждого внешнего объекта было только одно определение. В нем обобщены средства инициализации: теперь можно инициализировать автоматические массивы и структуры. Улучшен также препроцессор Си. Он включает более широкий набор директив условной компиляции, предоставляет возможность из макроаргументов генерировать строки в кавычках, а кроме того. содержит более совершенный механизм управления процессом макрорасширения.

4.1 Основные сведения о функциях

Начнем с того, что сконструируем программу, печатающую те строки вводимого текста, в которых содержится некоторый "образец", заданный в виде строки символов. (Эта программа представляет

собой частный случай функции grep системы UNIX.) Рассмотрим пример: в результате поиска образца "ould" в строках текста

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to The Heart's Desire!
```

мы получим

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits — and then
Re-mould it nearer to the Heart's Desire!
```

Работа по поиску образца четко распадается на три этапа:

```
while (существует еще строка)
if (строка содержит образец)
напечатать ее
```

Хотя все три составляющие процесса поиска можно поместить в функцию *main*, все же лучше сохранить приведенную структуру и каждую ее часть реализовать в виде отдельной функции. Легче иметь дело с тремя небольшими частями, чем с одной большой, поскольку, если несущественные особенности реализации скрыты в функциях, вероятность их нежелательного воздействия друг на друга минимальна. Кроме того, оформленные в виде функций соответствующие части могут оказаться полезными и в других программах.

Конструкция "while (*существует еще строка*)" реализована в *getline* (см. главу 1), а фразу "*напечатать ее*" можно записать с помощью готовой функции *printf*. Таким образом, нам остается перевести на Си только то, что определяет, входит ли заданный образец в строку.

Чтобы решить эту задачу, мы напишем функцию *strindex(s,t)*, которая указывает место (индекс) в строке *s*, где начинается строка *t*, или -1, если *s* не содержит *t*. Так как в Си нумерация элементов в массивах начинается с нуля, отрицательное число -1 подходит в качестве признака неудачного поиска. Если далее нам потребуется более сложное отождествление по образцу, мы просто заменим *strindex* на другую функцию, оставив при этом остальную часть программы без изменений. (Библиотечная функция **strstr** аналогична функции *strindex* и отличается от последней только тем, что возвращает не индекс, а указатель.)

После такого проектирования программы ее "деталировка" оказывается очевидной. Мы имеем представление о программе в целом и знаем, как взаимодействуют ее части. В нашей программе образец для поиска задается строкой-литералом, что снижает ее универсальность. В главе 5 мы еще вернемся к проблеме инициализации символьных массивов и покажем, как образец сделать параметром, устанавливаемым при запуске программы. Здесь приведена несколько измененная версия функции *getline*, и было бы поучительно сравнить ее с версией, рассмотренной в главе 1.

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

int getline(char line[], int max);
int strindex(char source[], char searchfor[]);

char pattern[] ="ould"; /* образец для поиска */

/* найти все строки, содержащие образец */

main()
{
    char line[MAXLINE];
    int found = 0;
    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) >= 0) {
                printf ("%s", line);
                found++;
        }
    return found;
}

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

```
    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n') /* I.B.: misprint was here -lim instead of --lim */
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}

/* strindex: вычисляет место t в s или выдает -1, если t нет в s */
int strindex (char s[], char t[])
{
    int i, j, k;
    for (i = 0; s[i] != '\0'; i++) {
        for (j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++)
            ;
        if (k > 0 && t[k] == '\0')
            return i;
    }
    return –1;
}
```

Определение любой функции имеет следующий вид:

```
тип-результата имя-функции (объявления аргументов)
{
    объявления и инструкции
}
```

Отдельные части определения могут отсутствовать, как, например, в определении "минимальной" функции

```
dummy() { }
```

которая ничего не вычисляет и ничего не возвращает. Такая ничего не делающая функция в процессе разработки программы бывает полезна в качестве "хранителя места". Если тип результата опущен, то предполагается, что функция возвращает значение типа *int*.

Любая программа - это просто совокупность определений переменных и функций. Связи между функциями осуществляются через аргументы, возвращаемые значения и внешние переменные. В исходном файле функции могут располагаться в любом порядке; исходную программу можно разбивать на любое число файлов, но так, чтобы ни одна из функций не оказалась разрезанной.

Инструкция **return** реализует механизм возврата результата от вызываемой функции к вызывающей. За словом *return* может следовать любое выражение:

```
return выражение;
```

Если потребуется, *выражение* будет приведено к возвращаемому типу функции. Часто выражение заключают в скобки, но они не обязательны.

Вызывающая функция вправе проигнорировать возвращаемое значение. Более того, *выражение* в **return** может отсутствовать, и тогда вообще никакое значение не будет возвращено в вызывающую функцию. Управление возвращается в вызывающую функцию без результирующего значения также и в том случае, когда вычисления достигли "конца" (т. е. последней закрывающей фигурной скобки функции). Не запрещена (но должна вызывать настороженность) ситуация, когда в одной и той же функции одни **return** имеют при себе выражения, а другие - не имеют. Во всех случаях, когда функция "забыла" передать результат в **return**, она обязательно выдаст "мусор".

Функция *main* в программе поиска по образцу возвращает в качестве результата количество найденных строк. Это число доступно той среде, из которой данная программа была вызвана.

Механизмы компиляции и загрузки Си-программ, расположенных в нескольких исходных файлах, в разных системах могут различаться. В системе UNIX, например, эти работы выполняет упомянутая в главе 1 команда *cc*. Предположим, что три функции нашего последнего примера расположены в трех разных файлах: *main.c*, *getline.c* и *strindex.c*. Тогда команда

```
cc main.c getline.c strindex.c
```

скомпилирует указанные файлы, поместив результат компиляции в файлы объектных модулей *main.o*,

getline.o и *strindex.o*, и затем загрузит их в исполняемый файл *a.out*. Если обнаружилась ошибка, например в файле *main.c*, то его можно скомпилировать снова и результат загрузить ранее полученными объектными файлами, выполнив следующую команду:

```
cc main.c getline.o strindex.o
```

Команда *cc* использует стандартные расширения файлов ".c" и ".o", чтобы отличать исходные файлы от объектных.

Упражнение 4.1. Напишите функцию strindex(s, t), которая выдает позицию самого правого вхождения t в s или -1, если вхождения не обнаружено.

4.2 Функции, возвращающие нецелые значения

В предыдущих примерах функции либо вообще не возвращали результирующих значений (**void**), либо возвращали значения типа **int**. А как быть, когда результат функции должен иметь другой тип? Многие вычислительные функции, как, например, sqrt, sin и cos, возвращают значения типа double; другие специальные функции могут выдавать значения еще каких-то типов. Чтобы проиллюстрировать, каким образом функция может возвратить нецелое значение, напишем функцию *atof(s)*, которая переводит строку s в соответствующее число с плавающей точкой двойной точности. Функция *atof* представляет собой расширение функции *atoi*, две версии которой были рассмотрены в главах 2 и 3. Она имеет дело со знаком (которого может и не быть), с десятичной точкой, а также с целой и дробной частями, одна из которых может отсутствовать. Наша версия не является высококачественной программой преобразования вводимых чисел; такая программа потребовала бы заметно больше памяти. Функция *atof* входит в стандартную библиотеку программ: ее описание содержится в заголовочном файле <**stdlib.h**>.

Прежде всего отметим, что объявлять тип возвращаемого значения должна сама *atof*, так как этот тип не есть *int*. Указатель типа задается перед именем функции.

```
#include <ctype.h>
/*atof: преобразование строки s в double */
double atof (char s[])
{
    double val, power;
    int i, sign;

    for (i = 0; isspace(s[i]); i++)
        ;        /* игнорирование левых символов-разделителей */
    sign = (s[i] == '-') ? –1 : 1;
    if (s[i] =='+' || s[i] =='-')
        i++;
    for (val = 0.0; isdigit (s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]; i++) {
        val = 10.0 * val + (s.[i] - '0');
        power *= 10.0;
    }
    return sign * val / power;
}
```

Кроме того, важно, чтобы вызывающая программа знала, что *atof* возвращает нецелое значение. Один из способов обеспечить это - явно описать *atof* в вызывающей программе. Подобное описание демонстрируется ниже в программе простенького калькулятора (достаточного для проверки баланса чековой книжки), который каждую вводимую строку воспринимает как число, прибавляет его к текущей сумме и печатает ее новое значение.

```
#include <stdio.h>
#define MAXLINE 100
/* примитивный калькулятор */
main()
{
    double sum, atof (char[]);
    char line[MAXLINE];
    int getline (char line[], int max);

    sum = 0;
    while (getline(line, MAXLINE) > 0)
```

```
        printf ("\t%g\n", sum += atof(line));
    return 0;
}
```

В объявлении

```
double sum, atof (char[]);
```

говорится, что *sum* - переменная типа *double*, а *atof* - функция, которая принимает один аргумент типа *char[]* и возвращает результат типа *double*.

Объявление и определение функции *atof* должны соответствовать друг другу. Если в одном исходном файле сама функция *atof* и обращение к ней в *main* имеют разные типы, то это несоответствие будет зафиксировано компилятором как ошибка. Но если функция *atof* была скомпилирована отдельно (что более вероятно), то несоответствие типов не будет обнаружено, и *atof* возвратит значение типа *double*, которое функция *main* воспримет как *int*, что приведет к бессмысленному результату.

Это последнее утверждение, вероятно, вызовет у вас удивление, поскольку ранее говорилось о необходимости соответствия объявлений и определений. Причина несоответствия, возможно, будет следствием того, что вообще отсутствует прототип функции, и функция неявно объявляется при первом своем появлении в выражении, как, например, в

```
sum += atof(line);
```

Если в выражении встретилось имя, нигде ранее не объявленное, за которым следует открывающая скобка, то такое имя по контексту считается именем функции, возвращающей результат типа *int*; при этом относительно ее аргументов ничего не предполагается. Если в объявлении функции аргументы не указаны, как в

```
double atof();
```

то и в этом случае считается, что ничего об аргументах *atof* не известно, и все проверки на соответствие ее параметров будут выключены. Предполагается, что такая специальная интерпретация пустого списка позволит новым компиляторам транслировать старые Си-программы. Но в новых программах пользоваться этим - не очень хорошая идея. Если у функции есть аргументы, опишите их, если их нет, используйте слово *void*.

Располагая соответствующим образом описанной функцией *atof*, мы можем написать функцию *atoi*, преобразующую строку символов в целое значение, следующим образом:

```
/* atoi: преобразование строки s в int с помощью atof */
int atoi (char s[])
{
    double atof (char s[]);
    return (int) atof (s);
}
```

Обратите внимание на вид объявления и инструкции *return*. Значение выражения в

```
return выражение;
```

перед тем, как оно будет возвращено в качестве результата, приводится к типу функции. Следовательно, поскольку функция *atoi* возвращает значение *int*, результат вычисления *atof* типа *double* в инструкции *return* автоматически преобразуется в тип *int*. При преобразовании возможна потеря информации, и некоторые компиляторы предупреждают об этом. Оператор приведения явно указывает на необходимость преобразования типа и подавляет любое предупреждающее сообщение.

Упражнение 4.2. Дополните функцию *atof* таким образом, чтобы она справлялась с числами вида

```
123.45e-6
```

в которых после мантиссы может стоять e (или E) с последующим порядком (быть может, со знаком).

4.3 Внешние переменные

Программа на Си обычно оперирует с множеством внешних объектов: переменных и функций.

Прилагательное "внешний" (**external**) противоположно прилагательному "внутренний", которое относится к аргументам и переменным, определяемым внутри функций. Внешние переменные определяются вне функций и потенциально доступны для многих функций. Сами функции всегда являются внешними объектами, поскольку в Си запрещено определять функции внутри других функций. По умолчанию одинаковые внешние имена, используемые в разных файлах, относятся к одному и тому же внешнему объекту (функции). (В стандарте это называется *редактированием внешних связей (линкованием)* (*external linkage*).) В этом смысле внешние переменные похожи на области *COMMON* в Фортране и на переменные самого внешнего блока в Паскале. Позже мы покажем, как внешние функции и переменные сделать видимыми только внутри одного исходного файла.

Поскольку внешние переменные доступны всюду, их можно использовать в качестве связующих данных между функциями как альтернативу связей через аргументы и возвращаемые значения. Для любой функции внешняя переменная доступна по ее имени, если это имя было должным образом объявлено.

Если число переменных, совместно используемых функциями, велико, связи между последними через внешние переменные могут оказаться более удобными и эффективными, чем длинные списки аргументов. Но, как отмечалось в главе 1, к этому заявлению следует относиться критически, поскольку такая практика ухудшает структуру программы и приводит к слишком большому числу связей между функциями по данным.

Внешние переменные полезны, так как они имеют большую область действия и время жизни. Автоматические переменные существуют только внутри функции, они возникают в момент входа в функцию и исчезают при выходе из нее. Внешние переменные, напротив, существуют постоянно, так что их значения сохраняются и между обращениями к функциям. Таким образом, если двум функциям приходится пользоваться одними и теми же данными и ни одна из них не вызывает другую, то часто бывает удобно оформить эти общие данные в виде внешних переменных, а не передавать их в функцию и обратно через аргументы.

В связи с приведенными рассуждениями разберем пример. Поставим себе задачу написать программу-калькулятор, понимающую операторы +, -, * и /. Такой калькулятор легче будет написать, если ориентироваться на польскую, а не инфиксную запись выражений. (Обратная польская запись применяется в некоторых карманных калькуляторах и в таких языках, как Forth и Postscript.) В обратной польской записи каждый оператор следует за своими операндами. Выражение в инфиксной записи, скажем

```
(1 - 2) * (4 + 5)
```

в польской записи представляется как

```
1 2 – 4 5 + *
```

Скобки не нужны, неоднозначности в вычислениях не бывает, поскольку известно, сколько операндов требуется для каждого оператора.

Реализовать нашу программу весьма просто. Каждый операнд посылается в стек; если встречается оператор, то из стека берется соответствующее число операндов (в случае бинарных операторов два) и выполняется операция, после чего результат посылается в стек. В нашем примере числа 1 и 2 посылаются в стек, затем замещаются на их разность -1. Далее в стек посылаются числа 4 и 5, которые затем заменяются их суммой (9). Числа -1 и 9 заменяются в стеке их произведением (т. е. -9). Встретив символ новой строки, программа извлекает значение из стека и печатает его.

Таким образом, программа состоит из цикла, обрабатывающего на каждом своем шаге очередной встречаемый оператор или операнд:

```
while (следующий элемент не конец-файла)
    if (число)
         послать его в стек
    else if (оператор)
         взять из стека операнды
         выполнить операцию
         результат послать в стек
    else if (новая-строка)
```

```
        взять с вершины стека число и напечатать
    else
        ошибка
```

Операции "послать в стек" и "взять из стека" сами по себе тривиальны, однако по мере добавления к ним механизмов обнаружения и нейтрализации ошибок становятся достаточно длинными. Поэтому их лучше оформить в виде отдельных функций, чем повторять соответствующий код по всей программе. И конечно необходимо иметь отдельную функцию для получения очередного оператора или операнда.

Главный вопрос, который мы еще не рассмотрели, - это вопрос о том, где расположить стек и каким функциям разрешить к нему прямой доступ. Стек можно расположить в функции *main* и передавать сам стек и текущую позицию в нем в качестве аргументов функциям *push* ("послать в стек") и *pop* ("взять из стека"). Но функции *main* нет дела до переменных, относящихся к стеку, - ей нужны только операции по помещению чисел в стек и извлечению их оттуда. Поэтому мы решили стек и связанную с ним информацию хранить во внешних переменных, доступных для функций *push* и *pop*, но не доступных для *main*.

Переход от эскиза к программе достаточно легок. Если теперь программу представить как текст, расположенный в одном исходном файле, она будет иметь следующий вид:

```
#include /* могут быть в любом количестве */
#define  /* могут быть в любом количестве */


объявления функций для main


main() {...}
внешние переменные для push и pop

void push (double f) {...}
double pop (void) {...}

int getop(char s[]) {...}

подпрограммы, вызываемые функцией getop
```

Позже мы обсудим, как текст этой программы можно разбить на два или большее число файлов.

Функция *main* - это цикл, содержащий большой переключатель *switch*, передающий управление на ту или иную ветвь в зависимости от типа оператора или операнда. Здесь представлен более типичный случай применения переключателя *switch* по сравнению с рассмотренным в параграфе 3.4.

```
#include <stdio.h>
#include <stdlib.h> /* для atof() */

#define MAXOP 100  /* макс. размер операнда или оператора */
#define NUMBER '0' /* признак числа */

int getop (char []);
void push (double);
double pop (void);

/* калькулятор с обратной польской записью */
main()
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop (s)) != EOF) {
        switch (type) {
        case NUMBER:
            push (atof(s));
            break;
        case '+':
            push (pop() + pop());
            break;
        case '*':
            push (pop() * pop());
            break;
        case '-':
            op2 = pop();
            push (pop() - op2);
            break;
        case '/':
```

```
            pop2 = pop();
            if (op2 != 0.0)
                push (pop() / op2);
            else
                printf("ошибка: деление на нуль\n");
            break;
        case '\n':
            printf("\t%.8g\n", pop());
            break;
        default:
            printf("ошибка: неизвестная операция %s\n", s);
            break;
        }
    }
    return 0;
}
```

Так как операторы + и * коммутативны, порядок, в котором операнды берутся из стека, не важен, однако в случае операторов - и /, левый и правый операнды должны различаться. Так, в

```
push(pop() – pop()); /* НЕПРАВИЛЬНО */
```

очередность обращения к *pop* не определена. Чтобы гарантировать правильную очередность, необходимо первое значение из стека присвоить временной переменной, как это и сделано в main.

```
#define MAXVAL 100  /* максимальная глубина стека */

int sp = 0;         /* следующая свободная позиция в стеке */
double val[MAXVAL]; /* стек */

/* push: положить значение f в стек */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("ошибка: стек полон, %g не помещается\n", f);
}

/* pop: взять с вершины стека и выдать в качестве результата */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        printf ("ошибка: стек пуст\n");
    return 0.0;
    }
}
```

Переменная считается внешней, если она определена вне функции. Таким образом, стек и индекс стека, которые должны быть доступны и для *push*, и для *pop*, определяются вне этих функций. Но *main* не использует ни стек, ни позицию в стеке, и поэтому их представление может быть скрыто от *main*.

Займемся реализацией *getop* - функции, получающей следующий оператор или операнд. Нам предстоит решить довольно простую задачу. Более точно: требуется пропустить пробелы и табуляции; если следующий символ - не цифра и не десятичная точка, то нужно выдать его; в противном случае надо накопить строку цифр с десятичной точкой, если она есть, и выдать число NUMBER в качестве результата.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getop: получает следующий оператор или операнд */
int getop(char s[])
{
    int i, c;
    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0;
    if (!isdigit(c) && c != '.')
        return c;    /* не число */
    i = 0;
    if (isdigit(c)) /* накапливаем целую часть */
        while (isdigit(s[++i] == c = getch()))
            ;
    if (c =='.') /* накапливаем дробную часть */
    while (isdigit(s[++i] = c = getch()))
```

```
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}
```

Как работают функции *getch* и *ungetch*? Во многих случаях программа не может "сообразить", прочла ли она все, что требуется, пока не прочтет лишнего. Так, накопление числа производится до тех пор, пока не встретится символ, отличный от цифры. Но это означает, что программа прочла на один символ больше, чем нужно, и последний символ нельзя включать в число.

Эту проблему можно было бы решить при наличии обратной чтению операции "положить-назад", с помощью которой можно было бы вернуть ненужный символ. Тогда каждый раз, когда программа считает на один символ больше, чем требуется, эта операция возвращала бы его вводу, и остальная часть программы могла бы вести себя так, будто этот символ вовсе и не читался. К счастью, описанный механизм обратной посылки символа легко моделируется с помощью пары согласованных друг с другом функций, из которых *getch* поставляет очередной символ из ввода, а *ungetch* отправляет символ назад во входной поток, так что при следующем обращении к *getch* мы вновь его получим.

Нетрудно догадаться, как они работают вместе. Функция *ungetch* запоминает посылаемый назад символ в некотором буфере, представляющем собой массив символов, доступный для обеих этих функций; *getch* читает из буфера, если там что-то есть, или обращается к *getchar*, если буфер пустой. Следует предусмотреть индекс, указывающий на положение текущего символа в буфере.

Так как функции *getch* и *ungetch* совместно используют буфер и индекс, значения последних должны между вызовами сохраняться. Поэтому буфер и индекс должны быть внешними по отношению к этим программам, и мы можем записать *getch*, *ungetch* и общие для них переменные в следующем виде:

```
#define BUFSIZE 100

char buf[BUFSIZE];    /* буфер для ungetch */
int bufp = 0;         /* след. свободная позиция в буфере */

int getch(void)       /* взять (возможно возвращенный) символ */
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c)   /* вернуть символ на ввод */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: слишком много символов\n");
    else
        buf[bufp++] = c;
}
```

Стандартная библиотека включает функцию **ungetc**, обеспечивающую возврат одного символа (см. главу 7). Мы же, чтобы проиллюстрировать более общий подход, для запоминания возвращаемых символов использовали массив.

Упражнение 4.3. Исходя из предложенной нами схемы, дополните программу- калькулятор таким образом, чтобы она "понимала" оператор получения остатка от деления (%) и отрицательные числа.

Упражнение 4.4. Добавьте команды, с помощью которых можно было бы печатать верхний элемент стека (с сохранением его в стеке), дублировать его в стеке, менять местами два верхних элемента стека. Введите команду очистки стека.

Упражнение 4.5. Предусмотрите возможность использования в программе библиотечных функций sin, exp и pow. См. библиотеку <math.h> в приложении B (параграф 4).

Упражнение 4.6. Введите команды для работы с переменными (легко обеспечить до 26 переменных, каждая из которых имеет имя, представленное одной буквой латинского алфавита). Добавьте переменную, предназначенную для хранения самого последнего из напечатанных значений.

Упражнение 4.7. Напишите программу ungets(s), возвращающую строку s во входной поток. Должна ли ungets "знать" что-либо о переменных buf и bufp, или ей достаточно пользоваться только функцией ungetch?

Упражнение 4.8. Предположим, что число символов, возвращаемых назад, не превышает 1. Модифицируйте с учетом этого факта функции getch и ungetch.

Упражнение 4.9. В наших функциях не предусмотрена возможность возврата EOF. Подумайте, что надо сделать, чтобы можно было возвращать EOF, и скорректируйте соответственно программу.

Упражнение 4.10. В основу программы калькулятора можно положить применение функции getline, которая читает целиком строку; при этом отпадает необходимость в getch и ungetch. Напишите программу, реализующую этот подход.

4.4 Области видимости

Функции и внешние переменные, из которых состоит Си-программа, каждый раз компилировать все вместе нет никакой необходимости. Исходный текст можно хранить в нескольких файлах. Ранее скомпилированные программы можно загружать из библиотек. В связи с этим возникают следующие вопросы:

• Как писать объявления, чтобы на протяжении компиляции используемые переменные были должным образом объявлены?

• В каком порядке располагать объявления, чтобы во время загрузки все части программы оказались связаны нужным образом?

• Как организовать объявления, чтобы они имели лишь одну копию?

• Как инициализировать внешние переменные?

Начнем с того, что разобьем программу-калькулятор на несколько файлов. Конечно, эта программа слишком мала, чтобы ее стоило разбивать на файлы, однако разбиение нашей программы позволит продемонстрировать проблемы, возникающие в больших программах.

Областью видимости имени считается часть программы, в которой это имя можно использовать. Для автоматических переменных, объявленных в начале функции, областью видимости является функция, в которой они объявлены. Локальные переменные разных функций, имеющие, однако, одинаковые имена, никак не связаны друг с другом. То же утверждение справедливо и в отношении параметров функции, которые фактически являются локальными переменными.

Область действия внешней переменной или функции простирается от точки программы, где она объявлена, до конца файла, подлежащего компиляции. Например, если *main*, *sp*, *val*, *push* и *pop* определены в одном файле в указанном порядке, т. е.

```
main() {...}

int sp = 0;
double val[MAXVAL];

void push(double f) {...}
double pop(void) {...}
```

то к переменным *sp* и *val* можно адресоваться из *push* и *pop* просто по их именам; никаких дополнительных объявлений для этого не требуется. Заметим, что в *main* эти имена не видимы так же, как и сами *push* и *pop*.

Однако, если на внешнюю переменную нужно сослаться до того, как она определена, или если она определена в другом файле, то ее объявление должно быть помечено словом **extern**.

Важно отличать *объявление* внешней переменной от ее *определения*. Объявление объявляет свойства переменной (прежде всего ее тип), а определение, кроме того, приводит к выделению для нее памяти. Если строки

```
int sp;
double val[MAXVAL];
```

расположены вне всех функций, то они *определяют* внешние переменные *sp* и *val*, т. е. отводят для них память, и, кроме того, служат объявлениями для остальной части исходного файла. А вот строки

```
extern int sp;
extern double val[];
```

объявляют для оставшейся части файла, что *sp* - переменная типа *int*, а *val* - массив типа *double* (размер которого определен где-то в другом месте); при этом ни переменная, ни массив не создаются, и память им не отводится.

На всю совокупность файлов, из которых состоит исходная программа, для каждой внешней переменной должно быть одно-единственное *определение*; другие файлы, чтобы получить доступ к внешней переменной, должны иметь в себе объявление *extern*. (Впрочем, объявление *extern* можно поместить и в файл, в котором содержится определение.) В определениях массивов необходимо указывать их размеры, что в объявлениях *extern* не обязательно. Инициализировать внешнюю переменную можно только в определении. Хотя вряд ли стоит организовывать нашу программу таким образом, но мы определим push и pop в одном файле, а val и sp - в другом, где их и инициализируем. При этом для установления связей понадобятся такие определения и объявления:

В файле 1:

```
extern int sp;
extern double val[];

void push(double f) {...}
double pop(void) {...}
```

В файле2:

```
int sp = 0;
double val[MAXVAL];
```

Поскольку объявления *extern* находятся в начале *файла1* и вне определений функций, их действие распространяется на все функции, причем одного набора объявлений достаточно для всего *файла1*. Та же организация *extern*-объявлений необходима и в случае, когда программа состоит из одного файла, но определения sp и val расположены после их использования.

4.5 Заголовочные файлы

Теперь представим себе, что компоненты программы-калькулятора имеют существенно большие размеры, и зададимся вопросом, как в этом случае распределить их по нескольким файлам. Программу *main* поместим в файл, который мы назовем *main.c*; *push*, *pop* и их переменные расположим во втором файле, *stack.c*; а *getop* - в третьем, *getop.c*. Наконец, *getch* и *ungetch* разместим в четвертом файле *getch.c*; мы отделили их от остальных функций, поскольку в реальной программе они будут получены из заранее скомпилированной библиотеки.

Существует еще один момент, о котором следует предупредить читателя, - определения и объявления совместно используются несколькими файлами. Мы бы хотели, насколько это возможно, централизовать эти объявления и определения так, чтобы для них существовала только одна копия. Тогда программу в процессе ее развития будет легче и исправлять, и поддерживать в нужном состоянии. Для этого общую информацию расположим в заголовочном файле *calc.h*, который будем по мере необходимости включать в другие файлы. (Строка #include описывается в параграфе 4.11) В результате получим программу, файловая структура которой показана ниже:

```
main.c:
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}

calc.h:
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char[]);
```

```
int getch(void);
void ungetch(int);

getop.c:
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop (){
    ...
}

getch.c:
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
intbufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}

stack.c:
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

Неизбежен компромисс между стремлением, чтобы каждый файл владел только той информацией, которая ему необходима для работы, и тем, что на практике иметь дело с большим количеством заголовочных файлов довольно трудно. Для программ, не превышающих некоторого среднего размера, вероятно, лучше всего иметь один заголовочный файл, в котором собраны вместе все объекты, каждый из которых используется в двух различных файлах; так мы здесь и поступили. Для программ больших размеров потребуется более сложная организация с большим числом заголовочных файлов.

4.6 Статические переменные

Переменные *sp* и *val* в файле *stack.c*, а также *buf* и *bufp* в *getch.c* находятся в личном пользовании функций этих файлов, и нет смысла открывать к ним доступ кому-либо еще. Указание **static**, примененное к внешней переменной или функции, ограничивает область видимости соответствующего объекта концом файла. Это способ скрыть имена. Так, переменные *buf* и *bufp* должны быть внешними, поскольку их совместно используют функции *getch* и *ungetch*, но их следует сделать невидимыми для "пользователей" функций *getch* и *ungetch*.

Статическая память специфицируется словом **static**, которое помещается перед обычным объявлением. Если рассматриваемые нами две функции и две переменные компилируются в одном файле, как в показанном ниже примере:

```
static char buf[BUFSIZE]; /* буфер для ungetch */
static int bufp = 0;      /* след. свободная позиция в buf */

int getch(void) {...}

void ungetch(int c) {...}
```

то никакая другая программа не будет иметь доступ ни к *buf*, ни к *bufp*, и этими именами можно свободно пользоваться в других файлах для совсем иных целей. Точно так же, помещая указание **static** перед объявлениями переменных *sp* и *val*, с которыми работают только *push* и *pop*, мы можем скрыть их от остальных функций.

Указание **static** чаще всего используется для переменных, но с равным успехом его можно применять и к функциям. Обычно имена функций глобальны и видимы из любого места программы. Если же функция помечена словом **static**, то ее имя становится невидимым вне файла, в котором она определена.

Объявление **static** можно использовать и для внутренних переменных. Как и автоматические переменные, внутренние статические переменные локальны в функциях, но в отличие от автоматических, они не возникают только на период работы функции, а существуют постоянно. Это значит, что внутренние статические переменные обеспечивают постоянное сохранение данных внутри функции.

Упражнение 4.11. Модифицируйте функцию *getop* так, чтобы отпала необходимость в функции *ungetch*. Подсказка: используйте внутреннюю статическую переменную.

4.7 Регистровые переменные

Объявление **register** сообщает компилятору, что данная переменная будет интенсивно использоваться. Идея состоит в том, чтобы переменные, объявленные **register**, разместить на регистрах машины, благодаря чему программа, возможно, станет более короткой и быстрой. Однако компилятор имеет право проигнорировать это указание. Объявление **register** выглядит следующим образом:

```
register int x;
register char c;
```

и т. д. Объявление **register** может применяться только к автоматическим переменным и к формальным параметрам функции. Для последних это выглядит так:

```
f(register unsigned m, register long n)
{
    register int i;
    ...
}
```

На практике существуют ограничения на регистровые переменные, что связано с возможностями аппаратуры. Располагаться в регистрах может лишь небольшое число переменных каждой функции, причем только определенных типов. Избыточные объявления **register** ни на что не влияют, так как игнорируются в отношении переменных, которым не хватило регистров или которые нельзя разместить на регистре. Кроме того, применительно к регистровой переменной независимо от того, выделен на самом деле для нее регистр или нет, не определено понятие адреса (см. главу 5). Конкретные ограничения на количество и типы регистровых переменных зависят от машины.

4.8 Блочная структура

Поскольку функции в Си нельзя определять внутри других функций, он не является языком, допускающим блочную структуру программы в том смысле, как это допускается в Паскале и подобных ему языках. Но переменные внутри функций можно определять в блочно-структурной манере. Объявления переменных (вместе с инициализацией) разрешено помещать не только в начале функции, но и после любой левой фигурной скобки, открывающей составную инструкцию. Переменная, описанная таким способом, "затеняет" переменные с тем же именем, расположенные в объемлющих блоках, и существует вплоть до соответствующей правой фигурной скобки. Например, в

```
if (n > 0) {
    int i; /* описание новой переменной i */
    for (i = 0; i < n; i++)
    ...
}
```

областью видимости переменной *i* является ветвь *if*, выполняемая при n>0; и эта переменная никакого отношения к любым *i*, расположенным вне данного блока, не имеет. Автоматические переменные, объявленные и инициализируемые в блоке, инициализируются каждый раз при входе в блок. Переменные *static* инициализируются только один раз при первом входе в блок.

Автоматические переменные и формальные параметры также "затеняют" внешние переменные и функции с теми же именами. Например, в

```
int x;
int y;
f(double x)
```

```
{
    double y;
}
```

x внутри функции *f* рассматривается как параметр типа *double*, в то время как вне *f* это внешняя переменная типа *int*. То же самое можно сказать и о переменной *y*.

С точки зрения стиля программирования, лучше не пользоваться одними и теми же именами для разных переменных, поскольку слишком велика возможность путаницы и появления ошибок.

4.9 Инициализация

Мы уже много раз упоминали об инициализации, но всегда лишь по случаю, в ходе обсуждения других вопросов. В этом параграфе мы суммируем все правила, определяющие инициализацию памяти различных классов.

При отсутствии явной инициализации для внешних и статических переменных гарантируется их обнуление; автоматические и регистровые переменные имеют неопределенные начальные значения ("мусор").

Скалярные переменные можно инициализировать в их определениях, помещая после имени знак = и соответствующее выражение:

```
int x = 1;
char squote = '\'';
long day = 1000L * 60L * 60L * 24L; /* день в миллисекундах */
```

Для внешних и статических переменных инициализирующие выражения должны быть константными, при этом инициализация осуществляется только один раз до начала выполнения программы. Инициализация автоматических и регистровых переменных выполняется каждый раз при входе в функцию или блок. Для таких переменных инициализирующее выражение - не обязательно константное. Это может быть любое выражение, использующее ранее определенные значения, включая даже и вызовы функции. Например, в программе бинарного поиска, описанной в параграфе 3.3, инициализацию можно записать так:

```
int binsearch(int x, int v[], int n)
{
    int low = 0;
    int high = n-1;
    int mid;
}
```

а не так:

```
int low, high, mid;

low = 0;
high = n - 1;
```

В сущности, инициализация автоматической переменной - это более короткая запись инструкции присваивания. Какая запись предпочтительнее - в большой степени дело вкуса. До сих пор мы пользовались главным образом явными присваиваниями, поскольку инициализация в объявлениях менее заметна и дальше отстоит от места использования переменной.

Массив можно инициализировать в его определении с помощью заключенного в фигурные скобки списка инициализаторов, разделенных запятыми. Например, чтобы инициализировать массив *days*, элементы которого суть количества дней в каждом месяце, можно написать:

```
int days[] = {31, 28, 31, 30, 31, 30, 31. 31, 30, 31, 30, 31};
```

Если размер массива не указан, то длину массива компилятор вычисляет по числу заданных инициализаторов; в нашем случае их количество равно 12.

Если количество инициализаторов меньше числа, указанного в определении длины массива, то для внешних, статических и автоматических переменных оставшиеся элементы будут нулевыми. Задание слишком большого числа инициализаторов считается ошибкой. В языке нет возможности ни задавать

повторения инициализатора, ни инициализировать средние элементы массива без задания всех предшествующих значений. Инициализация символьных массивов - особый случай: вместо конструкции с фигурными скобками и запятыми можно использовать строку символов. Например, возможна такая запись:

```
char pattern[] = "ould";
```

представляющая собой более короткий эквивалент записи

```
char pattern[] = {'o', 'u', 'l', 'd', '\0'};
```

В данном случае размер массива равен пяти (четыре обычных символа и завершающий символ '\0').

4.10 Рекурсия

В Си допускается рекурсивное обращение к функциям, т. е. функция может обращаться сама к себе, прямо или косвенно. Рассмотрим печать числа в виде строки символов. Как мы упоминали ранее, цифры генерируются в обратном порядке - младшие цифры получаются раньше старших, а печататься они должны в правильной последовательности.

Проблему можно решить двумя способами. Первый - запомнить цифры в некотором массиве в том порядке, как они получались, а затем напечатать их в обратном порядке; так это и было сделано в функции *itoa*, рассмотренной в параграфе 3.6. Второй способ - воспользоваться рекурсией, при которой *printd* сначала вызывает себя, чтобы напечатать все старшие цифры, и затем печатает последнюю младшую цифру. Эта программа, как и предыдущий ее вариант, при использовании самого большого по модулю отрицательного числа работает неправильно.

```
#include <stdio.h>

/* printd: печатает n как целое десятичное число */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n / 10);
    putchar(n % 10 + '0');
}
```

Когда функция рекурсивно обращается сама к себе, каждое следующее обращение сопровождается получением ею нового полного набора автоматических переменных, независимых от предыдущих наборов. Так, в обращении *printd(123)* при первом вызове аргумент *n = 123*, при втором - *printd* получает аргумент 12, при третьем вызове - значение 1. Функция *printd* на третьем уровне вызова печатает 1 и возвращается на второй уровень, после чего печатает цифру 2 и возвращается на первый уровень. Здесь она печатает 3 и заканчивает работу.

Следующий хороший пример рекурсии - это быстрая сортировка, предложенная Ч.А.Р. Хоаром в 1962 г. Для заданного массива выбирается один элемент, который разбивает остальные элементы на два подмножества - те, что меньше, и те, что не меньше него. Та же процедура рекурсивно применяется и к двум полученным подмножествам. Если в подмножестве менее двух элементов, то сортировать нечего, и рекурсия завершается.

Наша версия быстрой сортировки, разумеется, не самая быстрая среди всех возможных, но зато одна из самых простых. В качестве делящего элемента мы используем серединный элемент.

```
/* qsort: сортирует v[left]...v[right] по возрастанию */
void qsort(int v[], int left, int right)
{
    int i, last;
    void swap(int v[], int i, int j);

    if (left >= right)     /* ничего не делается, если */
        return;            /* в массиве менее двух элементов */
    swap(v, left, (left + right)/2);   /* делящий элемент */
    last = left;                       /* переносится в v[0] */
    for(i = left+1; i <= right; i++)   /* деление на части */
```

```
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last);   /* перезапоминаем делящий элемент */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

В нашей программе операция перестановки оформлена в виде отдельной функции (*swap*), поскольку встречается в *qsort* трижды.

```
/* swap: поменять местами v[i] и v[j] */
void swap(int v[], int i, int j)
{
    int temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Стандартная библиотека имеет функцию **qsort**, позволяющую сортировать объекты любого типа.

Рекурсивная программа не обеспечивает ни экономии памяти, поскольку требуется где-то поддерживать стек значений, подлежащих обработке, ни быстродействия; но по сравнению со своим нерекурсивным эквивалентом она часто короче, а часто намного легче для написания и понимания. Такого рода программы особенно удобны для обработки рекурсивно определяемых структур данных вроде деревьев; с хорошим примером на эту тему вы познакомитесь в параграфе 6.5.

Упражнение 4.12. Примените идеи, которые мы использовали в *printd*, для написания рекурсивной версии функции *itoa*; иначе говоря, преобразуйте целое число в строку цифр с помощью рекурсивной программы.

Упражнение 4.13. Напишите рекурсивную версию функции *reverse(s)*, переставляющую элементы строки в ту же строку в обратном порядке.

4.11 Препроцессор языка Си

Некоторые возможности языка Си обеспечиваются препроцессором, который работает на первом шаге компиляции. Наиболее часто используются две возможности: **#include**, вставляющая содержимое некоторого файла во время компиляции, и **#define**, заменяющая одни текстовые последовательности на другие. В этом параграфе обсуждаются условная компиляция и макроподстановка с аргументами.

4.11.1 Включение файла

Средство *#include* позволяет, в частности, легко манипулировать наборами *#define* и объявлений. Любая строка вида

```
#include "имя-файла"
```

или

```
#include <имя-файла>
```

заменяется содержимым файла с именем *имя-файла*. Если *имя-файла* заключено в двойные кавычки, то, как правило, файл ищется среди исходных файлов программы; если такового не оказалось или имя-файла заключено в угловые скобки < и >, то поиск осуществляется по определенным в реализации правилам. Включаемый файл сам может содержать в себе строки *#include*.

Часто исходные файлы начинаются с нескольких строк *#include*, ссылающихся на общие инструкции *#define* и объявления *extern* или прототипы нужных библиотечных функций из заголовочных файлов вроде <*stdio.h*>. (Строго говоря, эти включения не обязательно являются файлами; технические детали того, как осуществляется доступ к заголовкам, зависят от конкретной реализации.)

Средство *#include* - хороший способ собрать вместе объявления большой программы. Он гарантирует, что все исходные файлы будут пользоваться одними и теми же определениями и объявлениями переменных, благодаря чему предотвращаются особенно неприятные ошибки. Естественно, при

внесении изменений во включаемый файл все зависимые от него файлы должны перекомпилироваться.

4.11.2 Макроподстановка

Определение макроподстановки имеет вид:

```
#define имя замещающий-текст
```

Макроподстановка используется для простейшей замены: во всех местах, где встречается лексема *имя*, вместо нее будет помещен *замещающий-текст*. Имена в *#define* задаются по тем же правилам, что и имена обычных переменных. Замещающий текст может быть произвольным. Обычно замещающий текст завершает строку, в которой расположено слово *#define*, но в длинных определениях его можно продолжить на следующих строках, поставив в конце каждой продолжаемой строки обратную наклонную черту \. Область видимости имени, определенного в *#define*, простирается от данного определения до конца файла. В определении макроподстановки могут фигурировать более ранние *#define*-определения. Подстановка осуществляется только для тех имен, которые расположены вне текстов, заключенных в кавычки. Например, если YES определено с помощью *#define*, то никакой подстановки в printf("YES") или в YESMAN выполнено не будет.

Любое имя можно определить с произвольным замещающим текстом. Например:

```
#define forever for( ; ; ) /* бесконечный цикл */
```

определяет новое слово *forever* для бесконечного цикла.

Макроподстановку можно определить с аргументами, вследствие чего замещающий текст будет варьироваться в зависимости от задаваемых параметров. Например, определим *max* следующим образом:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Хотя обращения к *max* выглядят как обычные обращения к функции, они будут вызывать только текстовую замену. Каждый формальный параметр (в данном случае A и B) будет заменяться соответствующим ему аргументом. Так, строка

```
x = max(p+q, r+s);
```

будет заменена на строку

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Поскольку аргументы допускают любой вид замены, указанное определение *max* подходит для данных любого типа, так что не нужно писать разные *max* для данных разных типов, как это было бы в случае задания с помощью функций.

Если вы внимательно проанализируете работу *max*, то обнаружите некоторые подводные камни. Выражения вычисляются дважды, и если они вызывают побочный эффект (из-за инкрементных операций или функций ввода-вывода), это может привести к нежелательным последствиям. Например,

```
max(i++, j++) /* НЕВЕРНО */
```

вызовет увеличение *i* и *j* дважды. Кроме того, следует позаботиться о скобках, чтобы обеспечить нужный порядок вычислений. Задумайтесь, что случится, если при определении

```
#define square(x) x*x /* НЕВЕРНО */
```

вызвать *square (z+1)*.

Тем не менее макросредства имеют свои достоинства. Практическим примером их использования является частое применение *getchar* и *putchar* из <stdio.h>, реализованных с помощью макросов, чтобы из6ежать расходов времени от вызова функции на каждый обрабатываемый символ. Функции в <ctype.h> обычно также реализуются с помощью макросов. Действие *#define* можно отменить с помощью **#undef**:

```
#undef getchar
int getchar(void) {...}
```

Как правило, это делается, чтобы заменить макроопределение настоящей функцией с тем же именем.

Имена формальных параметров не заменяются, если встречаются в заключенных в кавычки строках. Однако, если в замещающем тексте перед формальным параметром стоит знак #, этот параметр будет заменен на аргумент, заключенный в кавычки. Это может сочетаться с конкатенацией (склеиванием) строк, например, чтобы создать макрос отладочного вывода:

```
#define dprint(expr) printf(#expr " = %g\n", expr)
```

Обращение к

```
dprint(x/y);
```

развернется в

```
printf("x/y" " = %g\n", x/y);
```

а в результате конкатенации двух соседних строк получим

```
printf("x/y=%g\n", x/y);
```

Внутри фактического аргумента каждый знак " заменяется на \", а каждая \ на \\, так что результат подстановки приводит к правильной символьной константе.

Оператор ## позволяет в макрорасширениях конкатенировать аргументы. Если в замещающем тексте параметр соседствует с ##, то он заменяется соответствующим ему аргументом, а оператор ## и окружающие его символы-разделители выбрасываются. Например, в макроопределении *paste* конкатенируются два аргумента

```
#define paste(front, back) front ## back
```

так что *paste(name, 1)* сгенерирует имя *name1*.

Правила вложенных использований оператора ## не определены; другие подробности, относящиеся к ##, можно найти в приложении А.

Упражнение 4.14. Определите swap(t,x,y) в виде макроса, который осуществляет обмен значениями указанного типа t между аргументами x и y. (Примените блочную структуру.)

4.11.3 Условная компиляция

Самим ходом препроцессирования можно управлять с помощью условных инструкций. Они представляют собой средство для выборочного включения того или иного текста программы в зависимости от значения условия, вычисляемого вовремя компиляции.

Вычисляется константное целое выражение, заданное в строке **#if**. Это выражение не должно содержать ни одного оператора **sizeof** или приведения к типу и ни одной **enum**-константы. Если оно имеет ненулевое значение, то будут включены все последующие строки вплоть до **#endif**, или **#elif**, или **#else**. (Инструкция препроцессора **#elif** похожа на else if.) Выражение **defined**(*имя*) в **#if** есть 1, если *имя* было определено, и 0 в противном случае.

Например, чтобы застраховаться от повторного включения заголовочного файла *hdr.h*, его можно оформить следующим образом:

```
#if !defined(HDR)
#define HDR

/* здесь содержимое hdr.h */

#endif
```

При первом включении файла *hdr.h* будет определено имя *HDR*, а при последующих включениях препроцессор обнаружит, что имя *HDR* уже определено, и перескочит сразу на *#endif*. Этот прием может

оказаться полезным, когда нужно избежать многократного включения одного и того же файла. Если им пользоваться систематически, то в результате каждый заголовочный файл будет сам включать заголовочные файлы, от которых он зависит, освободив от этого занятия пользователя.

Вот пример цепочки проверок имени *SYSTEM*, позволяющей выбрать нужный файл для включения:

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
#include HDR
```

Инструкции **#ifdef** и **#ifndef** специально предназначены для проверки того, определено или нет заданное в них имя. И следовательно, первый пример, приведенный выше для иллюстрации **#if**, можно записать и в таком виде:

```
#ifndef HDR
#define HDR

/* здесь содержимое hdr.h */

#endif
```

[Назад] [Содержание] [Вперед]

[Главная] [Гостевая]