Глава 4

Функции и структура программы

Функции не только помогают разбить большие вычислительные задачи на набор маленьких, но и позволяют программистам пользоваться разработками предшественников, а не начинать все заново. Хорошо написанные функции скрывают подробности своей работы от тех частей программы, которым их знать не положено, таким образом проясняя задачу в целом и облегчая процесс внесения исправлений и дополнений.

Язык C построен так, чтобы сделать использование функций максимально простым и эффективным. Обычно программы на C состоят из множества мелких функций, а не нескольких сравнительно больших. Программа может храниться в одном или нескольких файлах исходного кода, которые можно компилировать по отдельности, а затем собирать вместе, добавляя к ним предварительно скомпилированные функции из библиотек. Здесь этот процесс не будет освещаться подробно, поскольку его частности сильно меняются от системы к системе.

Объявление и определение функций — это тот раздел языка C, который претерпел самые заметные изменения с введением стандарта ANSI. В главе 1 говорилось о том, что теперь стало возможным объявлять типы аргументов при объявлении функции. Изменился также синтаксис определения функций с целью согласовать объявления с определениями. Благодаря этому компилятор может обнаружить гораздо больше ошибок, ранее ему недоступных. Более того, при правильном объявлении аргументов соответствующие приведения типов выполняются автоматически.

Сейчас стандарт задает более четкие правила области действия имен; в частности, выдвигается требование, чтобы каждый внешний объект определялся в программе ровно один раз. Более общей стала инициализация — теперь можно инициализировать автоматические массивы и структуры.

Усовершенствованию подвергся также и препроцессор C. В число новых возможностей препроцессора входят расширенный набор директив условной компиляции, новый метод создания символьных строк в кавычках из аргументов макросов, а также более совершенный контроль над процессом раскрытия макросов.

4.1. Основы создания функций

Для начала спроектируем и напишем программу, выводящую каждую из строк своих входных данных, в которой встречается определенный “шаблон” — заданная строка символов. (Это частный случай программы под названием `grep` из системы Unix.) Пусть, например, задан поиск строки `"ould"` в следующем наборе строк:

```
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![1]
```

На выходе программы получится следующее:

```
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!
```

Задача естественным образом раскладывается на три части:

```
while (на вход поступает очередная строка)
    if (строка содержит заданный шаблон)
        вывести ее
```

Хотя, разумеется, весь код для решения этих задач можно поместить в функцию `main`, лучше будет все-таки разделить его на части и организовать в виде отдельных функций, чтобы воспользоваться структурированностью программы. С маленькими частями удобнее иметь дело, чем с большим целым, поскольку в функциях можно скрыть несущественные детали реализации, при этом избежав риска нежелательного вмешательства одной части программы в другую. К тому же отдельные части могут даже оказаться полезными для других задач.

Первую часть нашего алгоритма (ввод строки) реализует функция `getline`, написанная еще в главе 1, а третью (вывод результата) — функция `printf`, которую давно написали за нас. Таким образом, нам осталось только написать функцию, которая бы определяла, содержит ли текущая строка заданный шаблон-образец.

Эта задача будет решена с помощью функции `strindex(s,t)`, которая возвращает позицию (или индекс) в строке `s`, с которой начинается строка `t`, либо `-1`, если строка `s` не содержит `t`. Поскольку массивы в C начинаются с нулевой позиции, индексы могут быть нулевыми или положительными, а отрицательное значение наподобие `-1` удобно для сигнализации ошибки. Если впоследствии понадобится применить более расширенный алгоритм поиска образцов в тексте, в программе достаточно будет заменить только `strindex`, а остальной код не надо будет изменять. (В стандартной библиотеке есть функция `strstr`, аналогичная по своему назначению `strindex`, только она возвращает не индекс, а указатель.)

Имея перед собой общую схему алгоритма, запрограммировать его детали уже не так сложно. Ниже приведен полный текст программы, по которому можно судить, как все его части сведены воедино. Образец, который можно разыскивать в тексте, пока что представляется литеральной строкой, что никак нельзя считать достаточно общим механизмом реализации. Вскоре будет рассмотрен вопрос инициализации символьных массивов, а в главе 5 рассказывается, как сделать задаваемый образец параметром, который бы указывался при запуске программы. В программу также включена несколько модифицированная версия функции `getline`; возможно, вам будет поучительно сравнить ее с приведенной в главе 1.

```
#include <stdio.h>
#define MAXLINE 1000    /* максимальная длина входной строки */

int getline(char line[], int max);
```

[1] Когда к жизни Любовь меня в мир призвала,
Мне уроки любви она сразу дала,
Ключ волшебный сковала из сердца частичек
И к сокровищам духа меня привела. (*Омар Хайям, пер. Н. Тенигиной*)

```
int strindex(char source[], char searchfor[]);

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

/* поиск всех строк, содержащих заданный 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: считывает строку в s, возвращает ее длину */
int getline(char s[], int lim)
{
    int c, i;

    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}

/* strindex: возвращает индекс строки t в s, -1 при отсутствии */
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;
}
```

Каждое из определений функций имеет следующую форму:

тип-возвращ-знач имя-функции(*объявления аргументов*)
{
    объявления и операторы
}

Различные части этого определения могут отсутствовать; самая минимальная функция имеет вид

```
dummy() {}
```

Она ничего не делает и ничего не возвращает. Такого рода функция, не выполняющая никаких операций, часто бывает полезна как заменитель (“заглушка”) в ходе разработки программы. Если тип возвращаемого значения опущен, по умолчанию подразумевается `int`.

Всякая программа является всего лишь набором определений переменных и функций. Функции обмениваются данными посредством передачи аргументов и возвращения значений, а также через внешние переменные. Функции могут следовать друг за другом в файле исходного кода в любом порядке, и текст программы можно разбивать на любое количество файлов, но при этом запрещено делить текст функции между файлами.

Оператор `return` — это механизм возвращения значений из вызываемой функции в вызывающую. После `return` может идти любое выражение:

```
return выражение
```

По мере необходимости *выражение* преобразуется в тип, возвращаемый функцией согласно ее объявлению и определению. Часто *выражение* заключают в круглые скобки, но это не обязательно.

Вызывающая функция имеет полное право игнорировать возвращаемое значение. Более того, после `return` вовсе не обязано стоять выражение; в этом случае в вызывающую функцию ничего не передается. Точно так же управление передается в вызывающую функцию без возврата значения, если достигнут конец тела функции, т.е. закрывающая правая фигурная скобка. Если функция возвращает значение из одного места и не возвращает из другого, это допускается синтаксисом языка, но может указывать на ошибку. Во всяком случае, если функция не возвращает значения, хотя по объявлению должна это делать, в передаваемых ею данных гарантированно будет содержаться “мусор” — случайные числа.

Программа поиска образцов в строках возвращает из `main` результат работы программы — количество найденных совпадений. Это число может использоваться операционной средой, которая вызвала программу.

Процедура компиляции и компоновки программы на C, хранящейся в нескольких файлах исходного кода, сильно отличается в разных системах. Например, в системе Unix все это делает команда `cc`, уже упоминавшаяся в главе 1. Предположим, что три разные функции программы хранятся в трех файлах: `main.c`, `getline.c` и `strindex.c`. Для компиляции и компоновки их в программу применяется следующая команда:

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

Она помещает объектный код, полученный в результате компиляции, в файлы `main.o`, `getline.o` и `strindex.o`, а затем компонует их все в выполняемую программу `a.out`. Если, скажем, в файле `main.c` встретилась синтаксическая ошибка, этот файл можно будет перекомпилировать заново и скомпоновать с уже существующими объектными файлами с помощью такой команды:

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

При использовании команды `cc` объектный код отличают от исходного благодаря применению разных расширений имен файлов — соответственно `.o` и `.c`.

Упражнение 4.1. Напишите функцию `strrindex(s,t)`, которая бы возвращала индекс самого правого вхождения строки `t` в `s`, либо `-1`, если такой строки в `s` нет.

4.2. Функции, возвращающие нецелые значения

До сих пор в наших примерах функции либо не возвращали никаких значений (`void`), либо возвращали числа типа `int`. А что если функция должна возвращать что-то другое? Многие функции для математических расчетов, такие как `sqrt`, `sin` или `cos`, возвращают числа типа `double`. Другие специализированные функции возвращают данные и других типов. Чтобы показать это на примере, напишем функцию `atof(s)` для преобразования строки символов `s` в вещественное число двойной точности, которое она изображает. Функция `atof` — это расширенная версия функции `atoi`, разные версии которой демонстрировались в главах 2 и 3. Она может обрабатывать знак и десятичную точку, а также различать, присутствует ли в числе целая и/или дробная часть. Наша версия не является высококлассной функцией преобразования, потому что иначе она бы заняла намного больше места. В стандартной библиотеке C функция `atof` имеется и объявлена в заголовочном файле `<stdlib.h>`.

Во-первых, следует явным образом объявить тип возвращаемого из `atof` значения, поскольку это не `int`. Имя типа ставится перед именем функции:

```
#include <ctype.h>

/* atof:  преобразование строки s в число типа double */
double atof(char s[])
{
    double val, power;
    int i, sign;

    for(i = 0; isspace(s[i]); i++)  /* пропуск пробелов */
        ;
    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;
    }
    return sign * val / power;
}
```

Во-вторых, и это не менее важно, вызывающая функция должна знать, что `atof` возвращает нецелое число. Один из способов сделать это — объявить `atof` прямо в вызывающей функции. Такое объявление показано ниже в программе-калькуляторе — настолько примитивной, что с ее помощью едва ли можно подбить даже бухгалтерский баланс. Она считывает числа по одному в строке (перед числом может стоять знак), складывает их и выводит текущую сумму после ввода каждой строки:

```
#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 программе имеется следующее объявление:

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

В нем говорится, что `sum` является переменной типа `double`, а `atof` — функцией, принимающей один аргумент типа `char[]` и возвращающей значение типа `double`.

Функция `atof` должна иметь согласованные между собой объявление и определение. Если сама функция `atof` и ее вызов в функции `main` имеют несогласованные типы и находятся в одном файле исходного кода, то компилятор выдаст сообщение об ошибке. Однако если (что более вероятно) функция `atof` компилируется отдельно, то несоответствие типов замечено не будет — функция возвратит число типа `double`, а `main` воспримет его как `int`, и в результате получится бессмыслица.

В свете всего сказанного насчет соответствия объявлений и определений функций это может показаться странным. Но дело в том, что здесь отсутствует прототип функции и функция объявляется неявным образом — своим первым появлением в таком выражении, как

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

Если в выражении встречается имя, которое ранее не объявлялось, и если после него стоят круглые скобки, оно по контексту объявляется функцией. По умолчанию считается, что эта функция возвращает число типа `int`, а относительно ее аргументов не делается никаких предположений. Более того, если объявление функции не содержит аргументов, то и в этом случае не делается никаких предположений относительно их типа и количества, как в этом примере:

```
double atof();
```

Проверка соответствия параметров для `atof` просто отключается. Это особое значение пустого списка аргументов введено для того, чтобы старые программы на C могли компилироваться новыми компиляторами. Однако использовать этот стиль в новых программах — дурной тон. Если функция принимает аргументы, объявляйте их. Если же она не принимает никаких аргументов, пишите `void`.

При наличии функции `atof`, объявленной должным образом, с ее помощью можно написать функцию `atoi` (для преобразования строки в целое число):

```
/* atoi:  преобразует строку s в целое число с помощью atof */
int atoi(char s[])
{
```

```
    double atof(char s[]);

    return (int) atof(s);
}
```

Обратите внимание на структуру объявления и на оператор `return`, обычная форма которого такова:

`return` *выражение*;

Стоящее в операторе *выражение* всегда преобразуется к возвращаемому типу перед выполнением оператора. Поэтому значение `atof`, имеющее тип `double`, автоматически преобразуется в тип `int` в том случае, если оно фигурирует в этом операторе, поскольку функция `atoi` возвращает тип `int`. При выполнении этой операции возможна частичная потеря информации, поэтому некоторые компиляторы в таких случаях выдают предупреждения. А вот при наличии явного приведения типов компилятору становится ясно, что операция выполняется намеренно, так что он не выдает предупреждений.

Упражнение 4.2. Усовершенствуйте функцию `atof` так, чтобы она понимала экспоненциальную запись чисел вида

```
123.45e-6
```

В этой записи после вещественного числа может следовать символ `e` или `E`, а затем показатель степени — возможно, со знаком.

4.3. Внешние переменные

Программа на C обычно состоит из набора внешних объектов, являющихся либо переменными, либо функциями. Прилагательное "внешний" здесь употребляется как противоположность слову "внутренний", которое относится к аргументам и переменным, определенным внутри функций. Внешние переменные определяются вне каких бы то ни было функций и поэтому потенциально могут использоваться несколькими функциями. Сами по себе функции — всегда внешние, поскольку в C нельзя определить функцию внутри другой функции. По умолчанию внешние переменные и функции имеют то свойство, что все ссылки на них по одним и тем же именам — даже из функций, скомпилированных отдельно, — являются ссылками на один и тот же объект программы. (Стандарт называет это свойство *внешним связыванием — external linkage*). В этом отношении внешние переменные являются аналогом переменных блока `COMMON` в языке Fortran или переменных в самом внешнем блоке в программе на Pascal. Позже будет показано, как определить внешние переменные и функции, чтобы они были доступны (видимы) только в одном файле исходного кода.

Поскольку внешние переменные имеют глобальную область видимости, они представляют собой способ обмена данными между функциями, альтернативный передаче аргументов и возврату значений. Любая функция может обращаться к внешней переменной по имени, если это имя объявлено определенным образом.

Если функциям необходимо передавать друг другу большое количество элементов данных, то внешние переменные удобнее и эффективнее, чем длинные списки аргументов. Однако, как было указано в главе 1, здесь не все так однозначно, поскольку от этого страдает структурированность программы и возникает слишком сильная перекрестная зависимость функций друг от друга.

Внешние переменные также бывают полезны из-за их более широкой области действия и более длительного времени жизни. Автоматические переменные являются внутренними для функций; они создаются при входе в функцию и уничтожаются при выходе из нее. С другой стороны, внешние переменные существуют непрерывно и хранят свои значения от одного вызова функции до другого. Поэтому если двум функциям необходимо иметь общий набор данных, но при этом ни одна из них не вызывает другую, бывает удобнее поместить общие данные во внешние переменные, чем передавать их туда-сюда в виде аргументов.

Рассмотрим эти вопросы более подробно на объемном примере. Задача состоит в том, чтобы написать программу-калькулятор с набором операций `+`, `-`, `*` и `/`. Калькулятор будет использовать обратную польскую, или бесскобочную, запись ввиду простоты ее реализации вместо обычной инфиксной. (Обратная польская запись применяется в некоторых карманных калькуляторах, а также в языках программирования Forth и Postscript.)

В обратной польской записи каждый знак операции стоит после своих операндов. Возьмем, например, следующее выражение:

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

В обратной польской записи оно выглядит так:

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

Скобки здесь не нужны, поскольку запись расшифровывается однозначно, если известно, сколько операндов у каждой операции.

Реализация алгоритма проста. Каждый операнд помещается в стек; как только поступает знак операции, из стека извлекается нужное количество операндов (для двухместной операции — два), к ним применяется данная операция, и результат снова помещается в стек. В приведенном примере в стек вначале помещаются 1 и 2, а затем они заменяются их разностью (–1). Затем в стек помещаются 4 и 5, после чего они заменяются их суммой (9). Произведение –1 и 9, т.е. –9, заменяет оба вычисленных операнда в стеке. Как только встретился конец строки входного потока, из стека извлекается и выводится на экран верхнее значение.

Таким образом, программа организована в виде цикла, выполняющего операции над знаками и операндами по мере их поступления:

```
while (очередной знак или операнд - не конец файла)
    if (число)
        поместить операнд в стек
    else if (знак операции)
        извлечь операнд из стека
        выполнить операцию
        поместить результат в стек
    else if (конец строки)
        извлечь и вывести верх стека
    else
        ошибка
```

Операции помещения в стек и извлечения из него тривиальны, но после добавления проверки и исправления ошибок эти операции становятся достаточно объемными, чтобы поместить их в отдельные функции, а не повторять длинные фрагменты кода несколько раз. Необходимо также иметь отдельную функцию для извлечения следующего операнда или знака операции.

Осталось обсудить основное проектное решение, необходимое для разработки данной программы. Это вопрос о том, где разместить стек, т.е. какая из функций программы будет с ним работать. Один из возможных вариантов — держать его в функции `main`, передавая как его, так и текущую позицию в нем в функции, которые помещают и извлекают данные. Но функции `main` нет нужды знать что-либо о переменных, которые управляют стеком; ей нужно только уметь помещать и извлекать данные. Поэтому будем хранить стек и связанную с ним информацию во внешних переменных, к которым будут обращаться функции `push` и `pop`, но не функция `main`.

Превратить это словесное описание в код сравнительно просто. Если пока что ограничиться программой, состоящей из одного файла исходного кода, получится следующее:

```
#include-директивы
#define-директивы

объявления функций, используемых в main

main() { ... }

внешние переменные для функций push и pop

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

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

функции, вызываемые из getop
```

Ниже рассказывается, как все это можно распределить по двум или нескольким файлам исходного кода.

Функция `main` организована в виде цикла, содержащего объемистый оператор `switch` по типам операций и операндов. Это более типичный пример применения оператора `switch`, чем тот, который приводился в разделе 3.4.

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

#define MAXOP   100   /* максимальный размер операнда или знака */
#define NUMBER  '0'   /* сигнал, что обнаружено число */

#int getop();
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 '/':
            op2 = pop();
            if(op2 != 0.0)
                push(pop() / op2);
            else
                printf("error: zero divisor\n");
            break;
        case '\n':
            printf("\t%.8g\n", pop());
            break;
        default:
            printf("error: unknown command %s\n", s);
            break;
        }
    }
    return 0;
}
```

Поскольку сложение (+) и умножение (*) — коммутативные операции, порядок, в котором их аргументы извлекаются из стека, не имеет значения. А вот для вычитания (–) и деления (/) следует различать левый и правый операнды. Допустим, вычитание представлено следующим образом:

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

В этом случае порядок, в котором обрабатываются два вызова `pop()`, не определен. Чтобы гарантировать правильный порядок, необходимо извлечь первый операнд заранее во временную переменную, как это и сделано в функции `main`.

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

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

/* push:  помещает число f в стек операндов */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: stack full, can't push %g\n", f);
}


/* pop:  извлекает и возвращает верхнее число из стека */
double pop(void)
```

```
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("error: stack empty\n");
        return 0.0;
    }
}
```

Переменная является внешней, если она определена вне каких бы то ни было функций. Поэтому сам стек и его индекс, которые совместно используются функциями `push` и `pop`, определены вне этих функций. Но функция `main` сама не обращается ни к стеку, ни к его индексу непосредственно, поэтому представление стека можно от нее спрятать.

Теперь займемся реализацией функции `getop`, которая доставляет на обработку очередной операнд или знак операции. Ее задача проста. Сначала необходимо пропустить пробелы и тому подобные символы. Если очередной символ не является цифрой или десятичной точкой, возвратить его. В противном случае накопить строку цифр (возможно, с десятичной точкой) и возвратить `NUMBER` — сигнал о том, что на вход поступило число.

```
#include <ctype.h>

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

/* getop:  извлекает следующий операнд или знак операции */
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))   /* накопление целой части */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.')     /* накопление дробной части */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF);
        ungetch(c);
    return NUMBER;
}
```

Что такое `getch` и `ungetch`? Часто бывает, что программа не может определить, достаточно ли она прочитала данных, до тех пор, пока не прочитает их слишком много. Один из примеров — это накопление символов, составляющих число. Пока не встретится первый символ, не являющийся цифрой, ввод числа не завершен; но как только он встретится, он оказывается лишним, и программа не знает, как его обрабатывать.

Проблема была бы решена, если бы существовал способ вернуть назад ненужный символ. Тогда каждый раз, когда программа считывала бы на один символ больше, чем

нужно, она могла бы вернуть его в поток ввода, а остальные части программы продолжали бы работать с этим потоком так, как будто последней операции ввода не было вовсе. К счастью, возврат символа в поток довольно легко смоделировать, написав для этого пару специальных функций. Функция getch будет извлекать следующий символ из потока ввода, а функция ungetch — запоминать символы, возвращаемые в поток, чтобы при следующем вызове getch вначале вводились они, а потом уже те, которые действительно поступают из потока ввода.

Совместная работа этих функций организована просто. Функция ungetch помещает возвращенный символ в общий буфер — массив символов. Функция getch читает данные из буфера, если они там есть, или вызывает getchar, если буфер пустой. Необходимо также иметь индексную переменную, которая бы хранила позицию текущего символа в буфере.

Поскольку буфер и индекс совместно используются функциями getch и ungetch, а также должны сохранять свои значения между вызовами этих функций, их следует сделать внешними переменными. Запишем итоговый код для getch, ungetch и их общих переменных:

```
#define BUFSIZE 100

char buf[BUFSIZE];   /* буфер для ungetch */
int bufp = 0;        /* следующая свободная позиция в nbuf */

int getch(void) /* ввод символа (возможно, возвращенного в поток) */
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c) /* возвращение символа в поток ввода */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: too many characters\n");
    else
        buf[bufp++] = c;
}
```

Стандартная библиотека содержит функцию ungetc, которая возвращает в поток один символ; она будет рассматриваться в главе 7. Здесь же для иллюстрации более общего подхода в качестве буфера возврата используется целый массив, а не один символ.

Упражнение 4.3. При наличии базовой структуры программы усовершенствование калькулятора уже не представляет особых трудностей. Реализуйте операцию взятия остатка (%) и работу с отрицательными числами.

Упражнение 4.4. Добавьте в программу реализацию команд для вывода верхнего элемента стека без его удаления, для создания в стеке дубликата этого элемента и для обмена местами двух верхних элементов. Также реализуйте команду очистки стека.

Упражнение 4.5. Добавьте реализацию библиотечных математических функций sin, exp и pow. См. заголовочный файл <math.h> в приложении Б, раздел 4.

Упражнение 4.6. Добавьте команды для работы с переменными. (Можно использовать 26 различных переменных, если разрешить имена только из одной буквы.) Введите переменную, обозначающую последнее выведенное на экран число.

Упражнение 4.7. Напишите функцию `ungets(s)`, возвращающую в поток целую строку символов. Следует ли этой функции знать о существовании переменных `buf` и `bufp`, или ей достаточно вызывать `ungetch`?

Упражнение 4.8. Предположим, что в поток ввода никогда не будет возвращаться больше одного символа. Доработайте функции `getch` и `ungetch` соответственно.

Упражнение 4.9. Наши функции `getch` и `ungetch` не могут корректно обработать символ конца файла `EOF`, возвращенный в поток ввода. Подумайте, как эти функции должны реагировать на `EOF` в буфере, а затем реализуйте ваш замысел.

Упражнение 4.10. В качестве альтернативного подхода можно использовать функцию `getline` для считывания целых строк. Тогда `getch` и `ungetch` становятся ненужными. Перепишите программу-калькулятор с использованием этого подхода.

4.4. Область действия

Функции и внешние переменные, составляющие программу на C, не обязательно компилируются в одно и то же время. Исходный текст программы может храниться в нескольких файлах, и к тому же могут подключаться заранее скомпилированные функции из библиотек. В этой связи возникает ряд вопросов, на которые необходимо знать ответы.

- Как записать объявления переменных, чтобы они правильно воспринимались во время компиляции?
- Как организовать объявления так, чтобы при компоновке программы все ее части правильно объединились в одно целое?
- Как организовать объявления так, чтобы каждая переменная присутствовала в одном экземпляре?
- Как инициализируются внешние переменные?

Рассмотрим эти вопросы на примере, распределив код программы-калькулятора по нескольким файлам. С практической точки зрения эта программа слишком мала, чтобы ее стоило так разбивать, но зато это хорошая иллюстрация важных приемов, применяемых в больших программах.

Областью действия (видимости) имени называется часть программы, в пределах которой можно использовать имя. Для автоматической переменной, объявляемой в начале функции, областью видимости является эта функция. Локальные переменные с одинаковыми именами, объявленные в разных функциях, не имеют никакого отношения друг к другу. То же самое справедливо и для параметров функций, которые по сути являются локальными переменными.

Область действия внешней переменной или функции распространяется от точки, в которой она объявлена, до конца компилируемого файла. Например, пусть `main`, `sp`, `val`, `push` и `pop` определены в одном файле, в порядке, показанном выше, т.е.

```
main() { ... }

int sp = 0;
double val[MAXVAL];

void push(double f) { ... }

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

Тогда переменные sp и val можно использовать в функциях push и pop, просто обращаясь по именам, — никаких дальнейших объявлений внутри функций не нужно. Однако эти переменные невидимы в функции main, как и собственно функции push и pop.

Если же необходимо обратиться к внешней переменной до ее определения или если она определена в другом файле исходного кода, то обязательно нужно вставить объявление с ключевым словом extern.

Важно понимать различие между *объявлением* внешней переменной и ее *определением*. Объявление сообщает, что переменная обладает определенными свойствами (в основном типом), а определение выделяет место в памяти для ее хранения. Если следующие строки фигурируют вне функций, то они являются *определениями* внешних переменных sp и val:

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

Благодаря этому определению выделяется память для хранения переменных. Кроме того, это еще и объявление, действительное до конца файла. С другой стороны, следующие строки дают только *объявление*, также действительное до конца файла, согласно которому sp имеет тип int, а val является массивом типа double (его размер определяется в другом месте). При этом память не выделяется и переменные не создаются.

Во всех файлах, образующих исходный текст программы, должно быть в общей сложности не больше одного *определения* внешней переменной; в других файлах могут содержаться *объявления* со словом extern, чтобы оттуда можно было к ней обращаться. (В файле, содержащем определение переменной, также могут находиться и extern-объявления ее же.) Размеры массивов обязательно указываются в определении, но необязательно — в объявлении со словом extern.

Инициализация внешней переменной выполняется только в определении.

Хотя в данной программе такая организация ни к чему, функции push и pop можно было бы определить в одном файле, а переменные val и sp — определить и инициализировать в другом. В итоге для связывания программы в единое целое понадобились бы следующие объявления и определения:

в файле ***file1***:

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

    void push(double f) { ... }

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

в файле ***file2***:

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

Поскольку объявления с extern в файле *file1* находятся впереди и вовне определений функций, они действительны во всех функциях; одного набора объявлений достаточно для всего файла. То же самое необходимо было бы записать, если бы определения переменных sp и val стояли после обращения к ним в одном и том же файле.

4.5. Заголовочные файлы

Теперь рассмотрим, как распределить программу-калькулятор по нескольким файлам исходного кода. Это необходимо было бы сделать, если бы каждый из компонентов программы имел намного большие размеры, чем сейчас. В таком случае функция `main` будет храниться в одном файле под именем `main.c`, функции `push` и `pop` вместе с их переменными — в другом файле, `stack.c`, а функция `getop` — в третьем файле, `getop.c`. Наконец, поместим функции `getch` и `ungetch` в четвертый файл, `getch.c`. Они будут отделены от остальных, потому что в настоящей, не учебной программе такие функции берутся из заранее скомпилированных библиотек.

Единственное, о чем стоит побеспокоиться, — это как разнести определения и объявления по разным файлам, сохранив возможность их совместного использования. Мы постарались сохранить централизованную структуру, насколько это было возможно, — собрать все необходимое в одном месте и использовать его на протяжении всей эволюции программы. Весь этот общий материал помещается в *заголовочный файл* `calc.h`, который подключается к файлам кода по мере необходимости. (Директива `#include` рассматривается в разделе 4.11.) В результате получается следующая программа:

calc.h:

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

main.c:

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

getop.c:

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

getch.c:

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 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) {
    ...
}
```

С одной стороны, хочется разрешить доступ из каждого файла только к самой необходимой информации; с другой стороны, уследить за множеством заголовочных файлов довольно нелегко. Нужно искать компромисс: добиться одного можно, только жертвуя другим. Пока объем программы не достиг некоторой средней величины, бывает удобнее хранить все, что необходимо для связи двух частей программы, в одном заголовочном файле. Именно такое решение было принято здесь. В программах, намного больших по объему, требуется более разветвленная организация и больше заголовочных файлов.

4.6. Статические переменные

Переменные `sp` и `val` в файле `stack.c`, а также `buf` и `bufp` в файле `getch.c` предназначены для внутреннего использования функциями в соответствующих файлах кода; всем остальным частям программы доступ к ним закрыт. Если объявление внешней переменной или функции содержит слово `static`, ее область действия ограничивается данным файлом исходного кода — от точки объявления до конца. Таким образом, внешние статические переменные — это механизм сокрытия имен наподобие `buf` и `bufp` в паре функций `getch-ungetch`, которые должны быть внешними, чтобы использоваться совместно, но не должны быть доступны за пределами указанных функций.

Статическое хранение переменной в памяти задается ключевым словом `static` в начале обычного объявления. Пусть в одном и том же файле компилируются две функции и две переменные:

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

int getch(void) { ... }

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

Тогда никакая другая функция не сможет обратиться к переменным `buf` и `bufp`, и не возникнет конфликта, если в других файлах программы будут употребляться такие же имена. Аналогично можно скрыть и переменные `sp` и `val`, объявив их статическими, чтобы только функции `push` и `pop` могли ими пользоваться для операций со стеком.

Чаще всего внешними статическими объявляются переменные, но такое объявление применимо и к функциям. Обычно имена функций являются глобальными и видимыми в любой части программы. Но если функцию объявить статической (`static`), ее имя будет невидимо за пределами файла, в котором она объявлена.

Объявление `static` применимо и к внутренним переменным. Внутренние статические переменные являются локальными по отношению к конкретной функции, как и автоматические. Но в отличие от автоматических статические переменные продолжают существовать непрерывно, а не создаются и уничтожаются при вызове и завершении функции. Получается, что внутренние статические переменные являются средством постоянного хранения скрытой информации внутри одной функции.

Упражнение 4.11. Измените функцию `getop` так, чтобы ей не нужно было использовать `ungetch`. *Подсказка:* воспользуйтесь внутренней статической переменной.

4.7. Регистровые переменные

Объявление с ключевым словом `register` сообщает компилятору, что соответствующая переменная будет интенсивно использоваться программой. Идея заключается в том, чтобы поместить такие (*регистровые*) переменные в регистры процессора и добиться повышения быстродействия и уменьшения объема кода. Впрочем, компилятор имеет право игнорировать эту информацию.

Объявления со словом `register` выглядят следующим образом:

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

Объявление `register` применимо только к автоматическим переменным и к формальным параметрам функций. В случае параметров оно выглядит так:

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

На практике существуют ограничения на употребление регистровых переменных, связанные со свойствами аппаратных устройств компьютера. В каждой функции только очень небольшое количество переменных (и только нескольких определенных типов) можно сделать регистровыми. Однако от лишних объявлений `register` не бывает никакого вреда, поскольку избыточные или неразрешенные объявления просто игнорируются компилятором. Не разрешается вычислять адрес регистровой переменной (эта тема рассматривается в главе 5), причем независимо от того, помещена ли она на самом деле в регистр процессора. Конкретные ограничения на количество и тип регистровых переменных зависят от системы и аппаратного обеспечения.

4.8. Блочная структура

Язык C не является блочно-структурным в том же смысле, как Pascal или другие языки, поскольку в нем функции нельзя определять внутри других функций. С другой стороны, внутри одной функции переменные можно определять в блочно-структурном стиле. Объявления переменных (в том числе инициализации) могут стоять после левой фигурной скобки, открывающей *любой* составной оператор, а не только после такой, которая открывает тело функции. Переменные, объявленные таким образом, блокируют действие любых переменных с теми же именами, объявленных во внешних блоках, и остаются в силе, пока не встретится соответствующая (закрывающая) правая скобка. Рассмотрим пример:

```
if (n > 0) {
    int i  /* объявляется новая переменная i */

    for (i = 0; i < n; i++)
        ...
}
```

В этом фрагменте кода область действия переменной `i` — та ветвь оператора `if`, которая соответствует "истине" в условии. Эта переменная не связана с любыми другими `i` за пределами блока. Автоматическая переменная объявляется и инициализируется в блоке всякий раз заново при входе в этот блок. Статическая переменная инициализируется только в первый раз при входе в блок.

Автоматические переменные, в том числе формальные параметры, также преобладают над внешними переменными и функциями с теми же именами, скрывая и блокируя действие их имен. Рассмотрим такое объявление:

```
int x;
int y;

f(double x)
{
    double y;
    ...
}
```

Внутри функции `f` любое употребление имени `x` будет относиться к параметру функции, имеющему тип `double`; за ее пределами `x` будет обозначать внешнюю переменную типа `int`. То же самое справедливо в отношении переменной `y`.

Стремясь программировать в хорошем стиле, лучше избегать имен переменных, которые перекрывают действие переменных из более внешних блоков, — слишком велика опасность путаницы и непредвиденных ошибок.

4.9. Инициализация

Инициализация мимоходом упоминалась уже много раз, но всегда как второстепенная тема по отношению к другим. Ранее уже рассматривались различные классы памяти для переменных, поэтому в данном разделе собраны некоторые правила инициализации, связанные с ними.

При отсутствии явной инициализации внешние и статические переменные гарантированно инициализируются нулями, а автоматические и регистровые получают неопределенные начальные значения ("мусор").

Скалярные переменные можно инициализировать прямо при объявлении, поставив после имени знак равенства и выражение:

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

Инициализирующие выражения для внешних и статических переменных должны быть константными; инициализация выполняется один раз, фактически до начала выполнения программы. А для автоматических и регистровых переменных инициализация выполняется каждый раз при входе в соответствующий блок.

Инициализирующее выражение автоматической или регистровой переменной не обязано быть константным — оно может содержать любые значения, определенные ранее, даже вызовы функций. Например, инициализацию в программе двоичного поиска из раздела 3.3 можно записать так:

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

Ранее она записывалась таким образом:

```
int low, high, mid;

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

Фактически инициализация автоматической переменной — это сокращенная запись для комбинации объявления и оператора присваивания. Какую из форм записи предпочесть — это, в общем-то, вопрос вкуса. Мы обычно употребляем явные операторы присваивания, поскольку инициализации в объявлениях труднее разглядеть и они дальше от точки использования той или иной переменной.

Чтобы инициализировать массив, необходимо поставить после его объявления и знака равенства список инициализирующих значений, разделенных запятыми, в фигурных скобках. Например, вот как инициализируется массив `days`, содержащий количество дней в каждом месяце года:

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

Если не указывать размер массива, компилятор вычислит его сам, сосчитав инициализирующие значения, которых в данном случае 12.

Если указано меньше инициализирующих значений, чем заданный размер массива, то недостающие значения будут заменены нулями для внешних, статических и автоматических переменных. Если же их слишком много, возникнет ошибка. Не существует способа задать повторение инициализирующих значений в сокращенном виде, а также инициализировать какой-нибудь средний элемент массива, не инициализировав все предыдущие.

Массивы символов — особый случай; для их инициализации можно использовать строку вместо фигурных скобок и запятых:

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

Эта запись короче, чем следующая, но эквивалентна ей:

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

В этом случае длина массива равна пяти (четыре символа плюс завершающий `'\0'`).

4.10. Рекурсия

Функции в C могут использоваться рекурсивно, т.е. функция может вызывать сама себя прямо или косвенно. Для примера рассмотрим задачу вывода числа в виде строки символов-цифр. Как уже говорилось, цифры генерируются в неправильном порядке: младшие цифры становятся известны раньше, чем старшие, однако выводить их надо как раз наоборот, начиная со старших.

У этой задачи есть два возможных решения. Одно заключается в том, чтобы помещать цифры в массив по мере генерирования, а затем вывести в обратном порядке. Это

делалось в функции `itoa` (см. раздел 3.6). Другой способ основан на рекурсии, при которой функция `printd` вначале вызывает сама себя для обработки старших (первых) цифр, а затем выводит самую последнюю. Эта функция также может не сработать с самым большим по модулю отрицательным числом.

```
#include <stdio.h>

/* printd:  вывод числа n десятичными цифрами */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n / 10);
    putchar(n % 10 + '0');
}
```

Когда функция рекурсивно вызывает сама себя, при каждом таком вызове создается новый набор автоматических переменных, независимый от предыдущего набора. Таким образом, при вызове `printd(123)` функция в первый раз получает аргумент `n = 123`, затем передает `12` во вторую копию `printd`, затем единицу — в третью копию, которая выводит эту единицу и возвращается на второй уровень. Там выводится двойка, и управление передается первому уровню, на котором выводится тройка и работа функции заканчивается.

Еще одним удачным примером рекурсии является быстрая сортировка — алгоритм сортировки, разработанный Ч.А.Р. Хоаром (C.A.R. Hoare) в 1962 г. Берется массив, выбирается один элемент, а все остальные делятся на два подмножества — элементы, бо́льшие выбранного или равные ему, и элементы, меньшие выбранного. Затем процедура применяется рекурсивно к каждому из двух подмножеств. Если в подмножестве меньше двух элементов, сортировка ему не требуется, и рекурсия на этом прекращается.

Наша версия быстрой сортировки — не самая быстрая из возможных, но зато одна из простейших. Для разбиения каждого подмножества используется его средний элемент.

```
/* qsort: сортировка v[left]...v[right] в порядке возрастания */
void qsort(int v[], int left, int right)
{
    int i, last;
    void swap(int v[], int i, int j);

    if (left >= right) /* ничего не делать, если в массиве */
        return;        /* меньше двух элементов */
    swap(v, left, (left+right)/2);   /* переместить */
    last = left;              /* разделитель в v[0] */
    for (i = left+1; i<= right; i++) /* упорядочение */
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last); /* вернуть разделитель на место */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Операция перемены элементов местами, `swap`, вынесена в отдельную функцию, поскольку она выполняется в `qsort` три раза.

```
/* swap:  обмен 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 включает версию `qsort`, которая может сортировать объекты любого типа.

Рекурсия может оказаться несовместимой с экономией памяти, поскольку нужно где-то хранить стек обрабатываемых чисел. Не дает она также и выигрыша в быстродействии. Но рекурсивный код обычно компактнее, и его значительно легче писать и дорабатывать, чем его аналог без рекурсии. Рекурсия особенно удобна при работе с рекурсивно определенными структурами данных наподобие деревьев; интересный пример будет рассмотрен в разделе 6.5.

Упражнение 4.12. Примените идеи, реализованные в `printd`, чтобы написать рекурсивную версию функции `itoa` — преобразование целого числа в строку путем рекурсивного вызова.

Упражнение 4.13. Напишите рекурсивную версию функции `reverse(s)`, которая обращает порядок символов в строке прямо на месте, без дополнительного строкового буфера.

4.11. Препроцессор C

Некоторые средства языка C реализованы с помощью препроцессора, работа которого по сути является первым этапом компиляции. Два наиболее часто используемых средства — это директива `#include`, включающая все содержимое заданного файла в компилируемый код, и директива `#define`, заменяющая некий идентификатор заданной последовательностью символов. Другие средства, описанные в этом разделе, — это условная компиляция и макросы с аргументами.

4.11.1. Включение файлов

С помощью включения файлов можно легко распоряжаться наборами директив `#define` и объявлений (среди прочего). Включение файлов задается одной из следующих строк в исходном коде:

```
#include "имя_файла"
#include <имя_файла>
```

При обработке текста строка заменяется содержимым файла *имя_файла*. Если *имя_файла* указано в кавычках, поиск файла обычно начинается с того места, где находится исходный текст программы. Если его там нет или если имя файла заключено в угловые скобки < и >, то поиск файла продолжается по правилам, зависящим от реализации языка. Включаемый файл может в свою очередь содержать директивы `#include`.

Очень часто в начале файла исходного кода стоит сразу несколько директив `#include`, которые подключают необходимые наборы директив `#define` и объявлений `extern` или же загружают объявления-прототипы библиотечных функций из таких заголовочных файлов, как `<stdio.h>`. (Строго говоря, они даже не обязаны быть файлами; способ обращения к заголовочным модулям зависит от системы и реализации языка.)

Директива `#include` — это оптимальный способ собрать все объявления в нужном месте в большой программе. Этот способ гарантирует, что все файлы исходного кода будут включать одни и те же макроопределения и объявления внешних переменных, таким образом ликвидируя возможность появления особенно неприятных ошибок. Разумеется, при внесении изменений во включаемый файл все файлы, которые ссылаются на него, должны быть перекомпилированы.

4.11.2. Макроподстановки

Следующая конструкция задает макроопределение, или макрос:

`#define` *`имя текст-для-замены`*

Это — макроопределение простейшего вида; всякий раз, когда *`имя`* встретится в тексте программы после этого определения, оно будет заменено на *`текст-для-замены`*. Имя в `#define` имеет ту же форму, что и имена переменных, а текст для замены может быть произвольным. Обычно текст для замены умещается на одной строке, но если он слишком длинный, его можно продлить на несколько строк, поставив в конце каждой строки символ продолжения (\). Область действия имени, заданного в директиве `#define`, распространяется от точки его определения до конца файла исходного кода. В макроопределении могут использоваться предыдущие определения. Подстановка выполняется только для идентификаторов и не распространяется на такие же строки символов в кавычках. Например, если для имени `YES` задано макроопределение, подстановка не выполняется для `printf("YES")` или `YESMAN`.

Текст для подстановки может быть совершенно произвольным. Например, таким образом можно определить имя `forever`, обозначающее бесконечный цикл:

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

Можно также определять макросы с аргументами, чтобы изменять текст подстановки в зависимости от способа вызова макроса. Например, так определяется макрос с именем `max`:

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

Хотя внешне он напоминает вызов функции, на самом деле `max` разворачивается прямо в тексте программы путем подстановки. Вместо формальных параметров (`A` или `B`) будут подставлены фактические аргументы, обнаруженные в тексте. Возьмем, например, следующий макровызов:

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

При компиляции он будет заменен на следующую строку:

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

Если всегда задавать при вызове соответствующие по типу аргументы, нет необходимости иметь несколько различных макросов `max` для различных типов данных, как это было бы с функциями.

Если внимательно изучить способ подстановки макроса `max`, можно заметить ряд "ловушек". Так, выражения в развернутой форме макроса вычисляются дважды, и это даст неправильный результат, если в них присутствуют побочные эффекты наподобие инкремента-декремента или ввода-вывода. Например, в следующем макровызове большее значение будет инкрементировано два раза:

```
max(i++, j++)   /* НЕПРАВИЛЬНО */
```

Необходимо также аккуратно использовать скобки, чтобы гарантировать правильный порядок вычислений. Представьте себе, что произойдет, если следующий макрос вызвать в виде `square(z+1)`:

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

И тем не менее макросы очень полезны. Один из практических примеров можно найти в файле `<stdio.h>`, в котором имена `getchar` и `putchar` часто определяются как макросы, чтобы избежать лишних вызовов функций для ввода отдельных символов.

Чтобы отменить определение имени, используется директива `#undef`. Обычно это делается для того, чтобы гарантировать, что имя функции действительно определяет функцию, а не макрос:

```
#undef getchar

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

Формальные параметры не заменяются аргументами, если они стоят в кавычках. Если же в тексте подстановки перед именем формального параметра стоит значок #, то эта комбинация заменяется строкой в кавычках, в которые вместо формального параметра подставляется аргумент. Эту возможность можно сочетать с конкатенацией строк, например для организации макроса отладочного вывода:

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

Вызовем его следующим образом:

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

Макрос будет развернут в следующую строку:

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

После конкатенации строк окончательный результат будет таким:

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

Внутри фактического аргумента (в строке) каждый символ `"` заменяется на `\"`, а каждый символ `\` на `\\`, так что в результате получается правильная строковая константа.

Операция препроцессора ## предоставляет возможность сцепить фактические аргументы в одну строку в процессе раскрытия макроса. Если параметр в подставляемом тексте находится рядом со знаком ##, этот параметр заменяется фактическим аргументом, сам знак ## и окружающие его пробелы удаляются, а результат снова анализируется препроцессором. Например, макрос `paste` сцепляет два своих аргумента:

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

Выражение `paste(name, 1)` порождает идентификатор `name1`.

Правила использования вложенных знаков ## загадочны и малопонятны; подробности можно найти в приложении А.

Упражнение 4.14. Определите макрос `swap(x,y)`, который обменивает местами значения двух аргументов типа `t`. (Примените блочную структуру.)

4.11.3. Условное включение

Существует возможность управлять самой препроцессорной обработкой, используя условные директивы, которые выполняются по ходу этой обработки. Это делает возможным условное включение фрагментов кода в программу в зависимости от условий, выполняющихся в момент компиляции.

В директиве `#if` выполняется анализ целочисленного выражения (которое не должно содержать операцию `sizeof`, приведение типов и константы, определенные через `enum`). Если выражение не равно нулю, в программу включаются все последующие строки вплоть до директивы `#endif`, `#elif` или `#else`. (Директива препроцессора `#elif` аналогична оператору `else if`.) Выражение `defined(`*имя*`)` в директиве `#if` равно единице, если *имя* определено с помощью `#define`, и нулю в противном случае.

Например, чтобы обеспечить включение файла `hdr.h` в текст программы не более одного раза, его содержимое следует окружить следующими условными директивами:

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

/* здесь находится содержимое файла hdr.h */

#endif
```

При первом включении `hdr.h` определяется имя `HDR`; при последующих включениях обнаруживается, что имя уже определено, и текст пропускается вплоть до `#endif`. Этот подход можно применить, чтобы избежать многократного включения набора связанных друг с другом файлов. Используя указанные директивы во всех файлах набора, можно смело включать в каждый файл все заголовочные модули, которые ему требуются, не думая о степени их взаимосвязанности.

В следующем фрагменте кода анализируется имя `SYSTEM` и принимается решение, какую версию заголовочного файла включать в программу:

```
#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
```

Директивы `#ifdef` и `#ifndef` — это специальные формы условной директивы для проверки того, определено то или иное имя или нет. Первый пример директивы `#if`, приведенный в начале раздела, можно было бы записать так:

```
#ifndef HDR
#define HDR

/* здесь находится содержимое файла hdr.h */

#endif
```