

Н. Джехани

Программирование
на
языке
Си

Программирование на языке Си

Narain Gehani

C: An Advanced Introduction

AT & T Bell Laboratories
Murray Hill, New Jersey

COMPUTER SCIENCE PRESS

Н. Джехани

Программирование на языке СИ

Перевод с английского
И. Г. ШЕСТАКОВА
под редакцией
Б. А. КУЗЬМИНА



Москва «Радио и связь»
1988

ББК 32.973
Д40
УДК 681.3.06

Редакция переводной литературы

Д40 **Джехани Н.**
Программирование на языке Си: Пер. с англ. - М.:
Радио и связь, 1988. - 272 с.: ил.
ISBN 5-256-00152-3.

Книга американского автора написана специально для программистов, хорошо владеющих одним из процедурных языков программирования, таких как Паскаль, ПЛ/1, Алгол 60, Симула 67, Алгол 68, Фортран или Ада. Особое внимание уделено наиболее сложным средствам языка Си. Рассмотрены стиль и дисциплина программирования, абстрагирование данных, препроцессор, инструментальные средства, применяемые при программировании на языке Си, параллельное программирование, операционная система UNIX. Приведено большое число примеров, взятых из реальных программ. В книгу включено приложение с ожидаемыми изменениями языка Си, обусловленными подготовкой стандарта языка.

Для широкого круга программистов.

Д $\frac{2405000000-044}{046(01)-88}$ 141-88

ББК 32.973

ISBN 5-256-00152-3 (рус.)
ISBN 0-88175-053-0 (англ.)

© 1985 Bell Telephone Laboratories, Inc.
© Перевод на русский язык, примечания
переводчика и редактора. Издательство
"Радио и связь", 1988.

ПРЕДИСЛОВИЕ

Язык программирования Си был разработан и реализован в 1972 году сотрудником фирмы AT&T Bell Laboratories Деннисом Ритчи. Хотя язык Си появился относительно недавно, популярность его росла очень быстро, и в настоящее время компиляторы этого языка созданы для многих машин, а список компиляторов продолжает быстро увеличиваться [31,70]. Растущая популярность языка Си обусловлена следующими двумя важными причинами. Во-первых, язык Си очень гибок: относительно просто его можно использовать в различных областях приложений. Во-вторых, большая часть программного обеспечения популярной операционной системы (ОС) UNIX написана на языке Си, который является основным языком программирования в этой системе.

Развитие языка Си продолжалось и после окончания его разработки и касалось, в частности, проверки типов данных и средств, облегчающих перенос программ в другую среду. Например, разработка проекта переноса ОС UNIX на компьютер Interdata 8/32 привела к некоторым добавлениям в язык Си, а именно, к включению в язык таких средств, как объединение (union), явное преобразование типа (cast) и определение типа (type definition) [6]. Позднее были сделаны попытки включения в язык Си средств абстрагирования данных [82]; в современной версии языка для абстрагирования данных имеются лишь ограниченные возможности. В настоящее время рассматривается проект стандарта ANSI языка Си; возможно, что процесс стандартизации приведет к дальнейшим изменениям языка Си, соответствующие предложения находятся сейчас на рассмотрении. Принятие стандарта ANSI языка Си намечено на конец 1985 года.¹ В качестве основы этой книги можно было бы выбрать предварительную версию стандарта ANSI языка Си. Однако такое решение не было принято, так как прежде чем версия ANSI языка Си станет стандартом, в нее, вероятно, будут внесены многочисленные изменения; кроме того, в настоящее время нет компиляторов, реализующих эту версию. В этой книге за основу взята версия языка Си, описанная в книге [77], являющейся последним вариантом этого справочного руководства по языку Си. Ожидаемые отличия излагаемой здесь версии языка Си от предварительной версии ANSI приведены в приложении В.

Язык Си предоставляет программисту большую свободу. Эта свобода - источник большой выразительности и один из главных источников силы языка Си, делающих его эффективным, универсальным и простым в использовании для различных областей применения. Однако бесконтрольное использование

¹ Стандарт ANSI языка не принят до сих пор.- *Прим. ред.*

этой свободы может привести к ошибкам, поэтому в этой книге предполагается, без потери общности, использование языка Си, основанное на определенной дисциплине программирования. Например, рассматривается ограниченный вариант оператора `switch`, для которого коды, соответствующие различным альтернативам, не перекрываются. Хотя в некоторых случаях с успехом могут быть использованы и перекрывающиеся коды альтернатив, такие коды трудны для понимания, модификации и сопровождения и являются потенциальным источником ошибок. Точно так же во всех иллюстративных программах книги все переменные инициализируются явно, не полагаясь на инициализацию по умолчанию, которая выполняется только для подмножества переменных.

Не существует компиляторов языка Си, которые проверяли бы все правила использования языка, предлагаемые в книге, и предупреждали бы о нарушениях дисциплины программирования. Однако многие такие нарушения обнаруживаются с помощью программы `lint` [3,4]; программисту настоятельно рекомендуется использовать программу `lint` для проверки программ перед их компиляцией и выполнением.

Язык Си является развивающимся языком - его возможности расширяются для удовлетворения осознанных потребностей, устраняются также его недостатки. С целью обеспечения совместимости с ранними версиями языка в нем сохранены некоторые устаревшие средства. Следовательно, некоторые средства в языке Си избыточны или малоупотребительны. За исключением особых случаев эти средства в книге не рассматриваются.

Эта книга специально предназначена для читателей с хорошим знанием по крайней мере одного из процедурных языков, например языка Паскаль, ПЛ/1, Алгол 60, Симула 67, Алгол 68, Фортран или Ада. В книге основное внимание уделено таким средствам языка Си, как описание типа данных, абстрагирование данных, особые ситуации, параллельное программирование, препроцессор языка Си и инструментальные средства, разработанные для использования совместно с программами на языке Си.

Некоторые из перечисленных средств языка Си требуют поддержки со стороны операционной системы, например ОС UNIX. Следовательно, возможность их использования может зависеть от конкретной операционной системы. При обсуждении системно-зависимых средств языка будет предполагаться, что программист разрабатывает программы на языке Си в ОС UNIX, поэтому будут также обсуждаться разнообразные возможности и инструментальные средства, доступные при программировании на языке Си в системе UNIX.

В этой книге приведено много примеров. Эти примеры принадлежат широкому спектру областей приложений, включая интерактивное программирование, системное программирова-

ние, приложения баз данных, обработку текстов и параллельное программирование. Все примеры прошли тестирование.¹ Многие из них взяты из реальных программ. Каждая глава заканчивается списком вопросов, дополняющих материал, представленный в этой главе.

В конце книги приведен список статей и книг по языку Си и смежным темам.

Для фрагментов программ на языке Си в книге используется шрифт со знаками одинаковой ширины ("машинописный"), например `return`; а для выделения формальных операций и синтаксических терминов - курсив, например стратегия *разделяй и властвуй*, печать сообщения об ошибке. Использование знаков постоянной ширины для фрагментов программ на языке Си согласуется со "стилем Си" [55].

Я благодарен фирме Bell Laboratories не только за предоставленную мне возможность написать эту книгу, но и за возможность более глубоко изучить язык Си. Я изучал язык Си, будучи сотрудником фирмы, а работа над книгой способствовала значительному углублению моего понимания языков программирования.

Я должен поблагодарить многих моих друзей за помощь при написании этой книги. Мне помогли А.В. Ахо, Р.Б. Аллен, М. Бьянки, Р.Л. Дрекслер, Дж. Фаррел, Дж.П. Фишберн, Д. Гей, Б.У. Керниган, Д. Линдерман, К. Маклокин, Д.А. Новитц, У.Д. Рум, А. Розлер, Б. Смит-Томас, Т.Г. Шимански и К.С. Уэдерелл. Боб Аллен прочел два варианта рукописи. Лэрри Розлер сообщил мне о предложениях по изменению языка Си, сделанных в процессе разработки для него стандарта ANSI.

Я высоко ценю также помощь Фреда Далримпла в вопросах параллельного программирования и Бьерна Страуструпа, рассказавшего мне о последней версии средств абстрагирования данных в языке Си++.

В течение нескольких последних лет Джон Линдерман и позднее Билл Рум обстоятельно отвечали на все мои многочисленные вопросы о языке Си и помогали мне лучше понять все его тонкости. Я благодарен им за эту помощь.

Марри-Хилл, шт. Нью-Йорк

Н. Джехани

¹ Эти примеры тестировались как в системе AT&T UNIX, версия 5.0 [3], так и в системе Berkeley UNIX [5]. Большинство программ без каких-либо изменений выполнялось в обеих операционных системах; однако в некоторых программах пришлось произвести незначительные изменения из-за различий в системах AT&T и Berkeley UNIX и их компиляторах языка Си. Различия в этих двух системах UNIX, влияющие на выполнение приведенных программ, и изменения, которые необходимо сделать в этих программах, указаны в соответствующих местах.

Глава 1

Введение и основные понятия

Язык Си был разработан в 1972 году Деннисом Ритчи для замены языка ассемблера при решении задач системного программирования в фирме Bell Laboratories. Успех применения языка превзошел все ожидания, и сейчас большая часть всего объема программирования в фирме Bell Laboratories (включая большую часть программного обеспечения ОС UNIX) выполняется на языке Си; более того, быстрыми темпами растет число пользователей Си за пределами фирмы.

1.1. ПРИМЕР ПРОГРАММЫ НА ЯЗЫКЕ СИ

Особенности программ, написанных на языке Си, можно продемонстрировать на примере небольшой программы, моделирующей простой калькулятор, который может складывать, умножать или делить. Данные, вводимые в эту программу, должны быть представлены строкой следующего формата:

A <оператор> B

где <оператор> может быть одним из знаков арифметических действий: +, -, * или /, а операнды A и B являются вещественными числами. Для упрощения программы пробелы между операндами и оператором не допускаются. Предполагается также, что единственной ошибкой, которая может быть совершена при использовании калькулятора, является задание знака оператора, не совпадающего ни с одним из четырех допустимых знаков.

Читатель, знакомый с языками высокого уровня, в состоянии понять работу программы калькулятора без особых затруднений. Идеи, положенные в основу программы, и языковые средства, использованные для ее написания, поясняются после текста программы; в этом разделе дается лишь краткое пояснение, а детальное обсуждение отложено до следующих глав.

```
/*-----*/  
/* main: простой калькулятор */  
/*-----*/
```

```

#include <stdio.h>

#define PROMPT ':'

main()
{
    float a, b;
    char opr;
    float result;

    while(getchar(PROMPT),scanf("%f%c%f",&a,&opr,&b)!=EOF)

        switch (opr) {
            case '+': result = a + b; break;
            case '-': result = a - b; break;
            case '*': result = a * b; break;
            case '/': result = a / b; break;
            default:
                printf("ОШИБКА *** неверен знак операции\n");
                exit(1);
        }

        printf("результат равен %g\n", result);
    }
    exit (0);
}

/*-----*/

```

Первые три строки программы - комментарии. Пара знаков "/*" обозначает начало комментария, а пара знаков "*/" - конец комментария.

Следующие две строки программы являются инструкциями препроцессора Си (все инструкции препроцессора начинаются со знака # в первой позиции строки). Первая инструкция

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

предписывает препроцессору заменить инструкцию *include* содержимым файла с именем `stdio.h`; этот файл содержит соответствующие описания средств ввода-вывода, обеспечиваемых стандартным библиотечным пакетом ввода-вывода `stdio`. Этот пакет находится в библиотеке стандартных программ `libc` (каждая программа на языке Си автоматически компилируется с библиотекой `libc`). Угловые скобки `<>` указывают, что файл `stdio.h` нужно искать в "стандартных местах" компьютерной системы.

В файле `stdio.h` содержится также описание константы `EOF`,¹ в котором ее значение часто указывается равным -1.

Вторая инструкция

```
#define PROMPT ':'
```

является указателем препроцессору языка Си на связь между символическим именем `PROMPT` и знаковой строкой `':'`, представляющей знак двоеточия; этот знак будет служить сигналом, приглашающим пользователя вводить данные. Вместо имени `PROMPT` препроцессор языка Си подставит его определение из правой части инструкции *define* (т.е. знаковую последовательность `':'`) во всех местах программы, где это имя встречается.

Программа калькулятора состоит из одной функции вида

```
main()
{
    ...
}
```

Слово `main` является именем этой функции, скобки без каких-либо знаков между ними указывают, что выполнение этой функции не требует параметров, тело функции заключено в фигурные скобки `{` и `}`. В системе UNIX выполнение программ, написанных на языке Си, начинается с выполнения функции с именем `main`, следовательно, функция с именем `main` должна быть составной частью любой законченной программы на языке Си.

Определения переменных

```
float a, b;
char opr;
float result;
```

задают переменные `a`, `b` и `result` как переменные с плавающей точкой, а переменную `opr` как знаковую переменную. Точка с запятой используется в качестве признака конца операторов, описаний и определений переменных.

Следующий оператор является циклом типа *while*, который в данном случае (без учета логически несущественных пробелов) имеет следующий вид:

¹ End Of File - конец файла.- *Прим. перев.*

```
while (exp != EOF) {
    последовательность операторов
}
```

Выполнение последовательности операторов внутри цикла типа *while* повторяется столько раз, сколько раз значение выражения *exp* не равно значению константы EOF. Выражение *exp* является составным выражением, сформированным из двух выражений

```
putchar(PROMPT)
```

и

```
scanf("%f%c%f", &a, &opr, &b)
```

с помощью оператора , (запятая). Значение выражения, сформированного с использованием оператора запятая, является значением его второго операнда; значение его первого операнда игнорируется. Например, значением выражения

```
putchar(PROMPT), scanf("%f%c%f", &a, &opr, &b)
```

является значение функции *scanf*; значение функции *putchar* игнорируется. Обе функции *putchar* и *scanf* включены в стандартный библиотечный пакет ввода-вывода *stdio*. Функция *scanf* соответствует операторам ввода по формату, имеющимся в языках, подобных языкам Фортран или ПЛ/1. В качестве аргументов она использует список форматов (например, %f, %c и %d), соответствующих списку переменных, значения которых должны быть введены, и список адресов этих переменных. Если при вводе обнаруживается конец данных, то функция *scanf* возвращает значение EOF; в остальных случаях она возвращает число элементов ввода, для которых установлено соответствие в списке форматов и адресов и значения которых присвоены соответствующим переменным.¹

Все аргументы в языке Си передаются по значению. Следовательно, передачу параметров по ссылке можно обеспечить передачей адресов переменных (например, указывая &a или &opr - оператор & дает адрес стоящей за ним в качестве операнда переменной). В языке Си имеются только функции, а подпрограммы (т.е. функции, которые могут не возвращать

¹ Более точно - см. приложение А.-Прим. ред.

значения в вызывающую программу) отсутствуют.¹ Каждая функция возвращает значение, даже если оно не нужно; если такая функция используется в качестве подпрограммы, ее значение просто игнорируется.

Нет никакой необходимости помещать вызов функции `putchar` в выражение в цикле *while*. Например, вышеприведенный цикл можно было бы написать и в таком виде:

```
putchar(PROMPT);
while (exp != EOF) {
    последовательность операторов
    putchar(PROMPT);
}
```

Однако в этом случае в тексте программы потребовалось бы дважды написать

```
putchar(PROMPT);
```

При необходимости выбора одной из нескольких взаимноисключающих возможностей используется оператор *switch*. Однако после завершения действия, соответствующего выбранному случаю, выполнение оператора *switch* не завершается, а продолжается до конца. Следовательно, необходим способ явного указания, когда завершить выполнение оператора *switch*. Один из таких способов - использование оператора *break* в качестве последнего оператора в группе операторов, соответствующих выбранному случаю. Выполнение оператора *break* приведет к завершению выполнения оператора *switch*. В операторе *switch*, используемом в программе калькулятора, возможны пять вариантов выбора:

```
switch (opr) {
    case '+': result = a + b; break;
    case '-': result = a - b; break;
    case '*': result = a * b; break;
    case '/': result = a / b; break;
    default:
        printf("ОШИБКА *** неверен знак операции\n");
        exit(1);
}
```

¹ Функция с значением типа `void` (пустой) является хорошим приближением для подпрограммы, не возвращающей значений (будет рассматриваться позднее). Тип `void` добавлен в язык Си позже, он не упоминается в книге [55], изданной в 1978 году, но уже включен в издание этой книги 1980 года [77].

Первые четыре случая рассчитаны на то, что переменная *opr* принимает одно из четырех знаковых значений: +, -, * или /. Пятый случай (default) предусмотрен для ситуации, в которой значение переменной не совпадает ни с одним из четырех уже указанных значений. В этом случае вызывается функция `printf`, выдающая сообщение об ошибке. Пара знаков `\n` обозначает специальный знак - признак перехода к следующей строке (newline). Знак `\` (backslash character) называется управляющим символом (*escape character*), который вместе со знаком (или числом, состоящим из одной, двух или трех восьмеричных цифр), следующим за ним, означает нечто специальное. Такая комбинация используется для обозначения знаков, кодам которых не соответствует какой-либо печатный знак. Вызов функции `exit`

```
exit(1);
```

обеспечивает завершение выполнения программы с кодом завершения, равным 1. В операционной системе UNIX принято, что ненулевое значение, возвращенное головной программой (программой с именем `main`), указывает на ее завершение по ошибке, а нулевое значение свидетельствует о нормальном, или успешном, завершении программы.

Программу калькулятора в том виде, в каком она приведена выше, нельзя отнести к "друзьям пользователя"; вместо попытки помочь пользователю в исправлении ошибок программа завершает работу после ввода пользователем неверного знака оператора. Можно сделать программу более удобной, заменив вызов функции `exit`:

```
exit(1);
```

операторами

```
printf("Допустимые операторы: +,-,* и /;");  
printf(" Попробуйте еще раз\n");  
continue;
```

Оператор *continue* обеспечивает продолжение выполнения программы с начала цикла *while*, где программа приглашает пользователя ввести данные еще раз.

За оператором *switch* следует вызов функции `printf`:

```
printf("Результат равен %g\n", result);
```

Результатом выполнения этой функции является печать строки

Результат равен *значение*

где *значение* - текущее значение переменной `result`; это значение печатается по формату `g` (он задается знаками `%g`), в соответствии с которым незначащие нули не печатаются, а десятичная точка печатается только в том случае, когда число не является целым.

И наконец, обнаружив, что входные данные введены полностью, программа нормально завершается вызовом функции `exit` с значением 0, сигнализирующим о том, что все в порядке:

```
exit(0);
```

Для завершения программы необязательно использовать функцию `exit`; работа программы может заканчиваться также выполнением всех операторов функции `main` или выполнением оператора `return` этой функции. Однако завершение программы с помощью функции `exit` делает информацию об успехе или неудаче доступной для других программ.

1.1.1. Компиляция и выполнение программы калькулятора в системе UNIX

После того как программа написана, ей предстоит пройти этапы компиляции и выполнения.

Предположим, что исходный текст программы калькулятора записан в файле с именем `calc.c` (в системе UNIX все файлы с исходными текстами программ на языке Си должны иметь суффикс `c`. Это соглашение, соблюдать которое необходимо для использования компилятора Си, облегчает применение таких средств, как `make`, при написании и сопровождении программ на языке Си; более подробно - см. приложение Б).

Для обнаружения в программах на языке Си ошибок некоторых типов может быть использована программа `lint`:

```
lint calc.c
```

После устранения ошибок, найденных с помощью программы `lint`, можно задать команду

```
cc calc.c
```

при выполнении которой обеспечивается компиляция программы (`cc` - имя компилятора) и ее связывание с библиотечными

функциями, используемыми в этой программе. Результатом компиляции при отсутствии ошибок является выполняемый файл с именем `a.out` (имя, присваиваемое по умолчанию), выполнить который можно просто командой

```
a.out
```

Имя `a.out` никак не указывает назначение программы; вызов компилятора с ключом `-o` позволяет программисту присвоить выполняемой версии программы подходящее имя:

```
cc -o calc calc.c
```

Приведем пример выполнения программы калькулятора:

```
$ calc
:59.0/4.0
результат равен 14.75
:39.0+44.0
результат равен 83
:$
```

Знак доллара `$` в системе UNIX является символом готовности,¹ указывающим, что система UNIX готова выполнить следующую команду пользователя. Работа программы прекращается после ввода пользователем символа `control-D` (в последней строке), обозначающего конец входных данных. Этот символ не имеет эквивалентного печатного знака и поэтому не показан. По существующему в системе UNIX соглашению символ `control-D` обозначает конец ввода данных или конец файла. Ниже приведен еще один пример выполнения программы, в котором выполнение заканчивается из-за неверно заданного знака операции:

```
$ calc
:2.0+37.5
результат равен 39.5
:5.0*4.5
результат равен 22.5
:5.0%4.5
ОШИБКА **** неверный знак операции
$
```

¹ На русский язык `prompt character` переводится также как "приглашение", "подсказка" или просто "промпт".- *Прим. ред.*

1.2. ОСНОВНЫЕ ПОНЯТИЯ

1.2.1. Алфавит

Алфавит (набор знаков) языка Си определяется реализацией языка Си. Например, для компьютеров DEC PDP-11, VAX-11¹ и AT&T 3B в качестве алфавита языка Си используется набор знаков кода ASCII, а для компьютеров IBM 370 - это набор знаков кода EBCDIC. В дальнейшем при обсуждении зависимых от реализации аспектов языка Си предполагается, если не указано явно, что используется компилятор Си в системе UNIX, установленной на компьютере семейства PDP-11 или VAX-11. В этой книге в качестве алфавита языка Си используется набор знаков кода ASCII.

Знаки пробела (blank), табуляции (tab) и перехода на следующую строку (newline) вместе с комментариями в общем случае будут обозначаться как *интервал* (*white space*).

1.2.2. Идентификаторы

Идентификаторы являются именами таких компонент программы, как переменные и функции. Имена могут состоять из любого числа знаков (букв, знаков подчеркивания "_" или цифр), но начинаться имена должны только с буквы или знака подчеркивания.² В языке Си буквы верхнего и нижнего регистров считаются различными [17]. Так же как и алфавит, правила образования идентификаторов зависят от реализации языка. Знаки как верхнего, так и нижнего регистров могут использоваться в реализациях языка Си на компьютерах PDP-11, VAX-11 и AT&T 3B, однако в некоторых реализациях используются знаки только верхнего регистра (знаки нижнего и верхнего регистров не различаются; знаки нижнего регистра преобразуются в знаки верхнего). Хотя идентификаторы могут быть любой длины, многие компиляторы языка Си различают идентификаторы только по первым восьми знакам. Например, идентификаторы

```
movement_detector  
movement_sensor
```

некоторыми компиляторами рассматриваются как идентичные из-за совпадения первых восьми знаков. Для идентификаторов

¹ PDP-11 и VAX-11 - торговые марки фирмы Digital Equipment Corporation.

² В системе UNIX, по соглашению, идентификаторы, начинающиеся со знака подчеркивания, резервируются за системными программами. Во избежание конфликтов не рекомендуется присваивать такие имена компонентам программы.

таких внешних программных компонент, как функции и внешние переменные (рассматриваемые ниже), могут быть установлены другие ограничения, зависящие от реализации языка.

Некоторые идентификаторы являются зарезервированными словами, которые называются *ключевыми*. Ключевые слова в программе должны использоваться только в соответствии с их предназначением. Список ключевых слов приведен ниже:

| | | | |
|----------|---------|----------|----------|
| auto | break | case | char |
| continue | default | do | double |
| else | entry | enum | extern |
| float | for | goto | if |
| int | long | register | return |
| short | sizeof | static | struct |
| switch | typedef | union | unsigned |
| void | while | | |

Ключевое слово *entry* в настоящее время не используется, оно зарезервировано для использования в будущем. В дополнение к вышеприведенному списку в некоторых реализациях языка Си применяются ключевые слова *fortran* и *asm*.

1.2.3. Литералы

*Литерал*¹ - это явное представление значения. Литералы в языке Си могут быть различных типов - *integer* (целые), *long integer* (длинные целые), *character* (знаковые), *floating point* (с плавающей точкой), *enumeration* (перечисляемые) или *string* (строки).

1.2.3.1. Литералы типа integer могут быть записаны в десятичной, восьмеричной или шестнадцатеричной системе. При записи в восьмеричной системе литералы должны начинаться с цифры 0, в шестнадцатеричной - с цифры 0 или буквы *x* (или *X*). Для обозначения шестнадцатеричных цифр от 10 до 15 могут использоваться буквы от *A* до *F* (или от *a* до *f*) соответственно. Десятичное число 12 в виде литерала типа *integer* можно записать как 12 (в десятичной системе), 014 (в восьмеричной системе) и 0XC (в шестнадцатеричной системе). Если такие литералы соответствуют значениям, превышающим диапазон представления чисел типа *integer*, то они считаются принадлежащими к типу *long integer*.

1.2.3.2. Литералы типа long integer задаются явно записью за целым литералом буквы *L* (или *l*).

¹ В терминологии языка Си литералы называются константами. Термин литерал в этой книге используется, чтобы отличать литералы от идентификаторов констант, обеспечиваемых применением препроцессора языка Си.

1.2.3.3. *Литералы типа character* формируются заключением одного знака между одиночными кавычками. Этот литерал может использоваться и как литерал типа integer с значением, которое является целочисленной интерпретацией двоичного представления литерала типа character. Некоторые знаки, не имеющие графического представления, такие как одиночная кавычка ' и обратная дробная черта \, обозначаются с помощью escape-последовательности так, как это показано в следующей таблице:¹

| Знак | Обозначение |
|---|-------------|
| Переход на следующую строку (newline) | \n |
| Горизонтальная табуляция (horizontal tab) | \t |
| Вертикальная табуляция (vertical tab) | \v |
| Шаг назад (backspace) | \b |
| Возврат каретки (carriage return) | \r |
| Смещение формы (form feed) | \f |
| Обратная дробная черта (backslash) | \\ |
| Одиночная кавычка (single quote) | \' |

Все вышеприведенные знаки могут быть также заданы как

\ddd

где через ddd обозначено восьмеричное число, состоящее из одной, двух или трех цифр и указывающее порядковый номер знака в таблице кодов ASCII (см. приложение Г). Например, вместо записи \n для обозначения знака перехода на следующую строку newline можно написать \012, запись \107 обозначает букву G, а запись \0 обозначает нулевой знак².

Ниже приведены примеры литералов типа character:

'a' знак а
 '\n' знак перехода на следующую строку (newline)
 '\\\' знак обратной дробной черты (backslash)
 '\'' знак одиночной кавычки
 '\107' буква G

¹ Последовательность из двух знаков \с, где знак с не является цифрой или одним из знаков n, t, v, b, r, f, \ и ', обозначает сам знак с.

² Символическое обозначение этого знака NUL, его числовое значение равно 0.

1.2.3.4. Литералы типа *floating point* задаются с помощью обычных обозначений:

24.0 2.4E1 (или 2.4e1) 240.0E-1

где число за буквой E (или e) указывает степень десяти, на которую умножается число, записанное слева от буквы E (или e). Предполагается, что все литералы типа *floating point* задают значения с двойной точностью.

1.2.3.5. *Перечисляемые литералы* являются идентификаторами, обозначаящими значения типа, определенного пользователем (см. гл.2).

1.2.3.6. *Литералы типа string* формируются заключением в двойные кавычки последовательности знаков (эта последовательность может не содержать ни одного знака). Например:

" . . . " "A"

"ошибка" "длинная строка"

Знак двойной кавычки может быть включен в литерал типа *string* с помощью знака *escape* \ , т.е. в любом месте, где знак " должен быть элементом литерала, нужно записать "\". С помощью знака *escape* литералы типа *string* могут быть продолжены на следующей строке текста программы:

"очень очень очень очень очень очень очень очень очень\
длинная строка"

Литералы типа *string* могут содержать также знаки, не имеющие графического представления, например, *newline* или *backspace*. В литерале

"первая строка\нвторая строка"

содержится знак *newline*, обозначенный \n. Если устройство печати следующий символ будет печатать в начале новой строки, то, будучи напечатанным, этот литерал будет иметь следующий вид:

первая строка
вторая строка

Литералы типа *string* являются фактически массивами знаков (см. подразд.2.2.1 и 2.2.6). Например, литерал типа *string*

эквивалентен массиву из 10 знаков

B, e, l, l, пробел, L, a, b, s, \0

По соглашению, строки (strings) в языке Си заканчиваются *нулевым* знаком \0. Обработка строк в языке Си основана на этом соглашении. В случае литералов типа string *нулевой* знак добавляется компилятором автоматически. Однако в случае строк, сформированных программистом явно, т.е. с использованием массива знаков, программист для выполнения вышеупомянутого соглашения должен добавить в конец строки нулевой знак.

1.2.4. Комментарии

Комментарии начинаются со знаков /* и заканчиваются знаками */. Комментарий может начинаться на одной строке текста программы и заканчиваться на другой строке. Вложенные комментарии не допускаются.

1.2.5. Точка с запятой - признак конца оператора

Знак точка с запятой используется в качестве признака конца описаний и операторов с одним исключением - точка с запятой не ставится после правой фигурной скобки } *составного* оператора, имеющего вид {...}. Использование в языке Си точки с запятой в качестве признака конца совпадает с использованием точки с запятой в языках ПЛ/1 и Ада и отличается в языке Паскаль, где этот знак играет роль разделителя операторов. Использование точки с запятой в качестве разделителя, возможно, является простым решением [30], однако экспериментально показано, что на практике такой подход чреват возникновением ошибок [23].

1.3. Константы

Определение констант обеспечивается препроцессором языка Си (рассматриваемым в гл. 9). Определения констант имеют следующий вид:

```
#define имя константы    литерал или имя константы  
#define имя константы    (выражения из констант)
```

С точки зрения препроцессора языка Си оба способа определения констант одинаковы. Они фактически являются макроопределениями вида

```
#define идентификатор строка замены
```

Такое определение приводит к замене в следующем за ним тексте программы последовательности знаков, составляющих *идентификатор*, последовательностью знаков, составляющих *строку замены* (такая замена выполняется при просмотре текста до тех пор, пока не встретится новое определение *идентификатора*). Следовательно, при использовании таких определений, как определения констант языка Си, нет необходимости в заключении выражения из констант в скобки; однако разумно это сделать во избежание неожиданных интерпретаций (более подробно - см. гл. 9).

Ниже в качестве примера приведены определения констант, использованных при определении структуры файла базы данных и задающих максимальный объем базы данных:

```
#define LN      20 /* длина имени + 1 для \0 */
#define LR      8 /* длина записи + 1 для \0 */
#define MAX_DB 100 /* макс. объем базы данных */
```

Еще один пример демонстрирует вторую форму представления оператора *define*:

```
#define EOF      (-1)
#define TOTAL_ELEM (M*N)
/* M и N - константы */
```

1.4. ЗАДАЧИ

1. Напишите программу калькулятора на привычном для Вас языке и сравните ее с версией этой же программы на языке Си. Как обозначаются неграфические знаки? Как записываются константы? Чем заканчиваются операторы?

2. Какие соглашения существуют в языках Паскаль и Ада для обозначения конца строки? Укажите все "за" и "против" явного указания концов строк с помощью *нулевого* знака.

3. Почему по сравнению со случаем использования точки с запятой в качестве признака окончания операторов возникновение ошибок более вероятно, если точка с запятой - разделитель операторов?

Глава 2

Типы и переменные

Тип - это множество значений плюс набор операций, которые могут быть выполнены с этими значениями [68]. *Переменной* называется компонента программы, используемая для хранения значения ассоциированного с ней типа. Присваивание переменной нового значения приводит к потере ее старого значения (если оно было присвоено).

Типы в языке Си делятся на две категории - *основные* и *производные*. К основным типам относятся знаковый (character), целый (integer), перечисляемый (enumeration), с плавающей точкой (floating point) и пустой (void). Типы character, integer, enumeration и floating point называются также *арифметическими*, так как они могут интерпретироваться как числа. Типы character, integer и enumeration называются также *целыми* (integral); тип floating point относится как к числам обычной, так и двойной точности. Производные типы конструируются из основных типов; ими являются массивы (array), функции (function), указатели (pointer), структуры (structure) и объединения (union).¹

Прежде чем перейти к детальному обсуждению типов в языке Си, уточним некоторые термины.

Объектом (object) называется область памяти.

Термин *описание* (declaration) объекта будет использоваться только для указания свойств объекта без назначения для него области памяти.

Термин *определение* (definition) будет использоваться для указания свойств объекта и назначения для него области памяти.²

Описания объектов нужны для того, чтобы обеспечить возможность ссылки на объекты, которые определяются позднее в файле, содержащем программу, или где-нибудь в других файлах, содержащих другие части программы. В процессе обсуждения типов иногда придется касаться описаний и определений переменных, имеющих тесную связь с типами. Здесь будут использоваться только простые формы описаний и определений, общую форму мы рассмотрим ниже.

¹ Производные типы в языке Ада означают нечто совсем другое.

² Важно помнить о различии между описанием и определением не только из-за частого использования этих терминов в дальнейшем, но и из-за того, что многие путают их, употребляя один термин в значении другого.

2.1. ОСНОВНЫЕ ТИПЫ

2.1.1. Знаки

Значения, ассоциированные с типом `char`, представляют собой элементы множества знаков, определенных реализацией языка. В качестве такого множества может выступать набор знаков кода ASCII. Например, оператором

```
char c, ch;
```

переменные `c` и `ch` определяются как знаковые.

Значения знаков хранятся как целые числа, что соответствует внутреннему представлению знаков. Следовательно, знаки можно рассматривать как целые числа и наоборот.¹ Такая двойственность удобна при программировании; некоторым функциям, в результате выполнения которых возвращаются знаковые значения, описанием присваивается тип `integer`, так что эти функции могут возвращать в вызывающую программу и целые значения, например `-1` (которое не соответствует никакому знаку), для индикации ошибки или конца файла.

Вследствие указанного соглашения следует соблюдать осторожность при использовании переменных типа `char`. Определяя переменные, служащие для хранения возвращаемых функциями знаковых значений, необходимо присваивать им тип `int`, чтобы избежать ошибок, когда возвращаемое функцией значение должно быть целым, например в особых ситуациях или при нарушении ограничений. Так, функция `getc` (из стандартного библиотечного пакета ввода `stdio`) предназначена для считывания следующего знака из стандартного файла ввода. Однако при обнаружении конца файла функция `getc`, подобно другим стандартным функциям, возвращает число `-1`, следовательно, `getc` определяется как функция, возвращающая значения типа `int` (а не `char`). Таким образом, переменные, используемые для хранения значений, возвращаемых функцией `getc`, должны быть определены как целые!

2.1.2. Целые

В зависимости от числа двоичных разрядов, отводимых для представления значений, целые могут быть типа `int`, `short int` (или просто `short`) и `long int` (или просто `long`). Например:

¹ Так как знаки могут интерпретироваться как целые числа и наоборот; компилятор не в состоянии обнаружить потенциальные ошибки, такие как неумышленное сложение знаковой и целой переменных.


```
int i, n;  
short int low, high;  
long int max;
```

Переменным типа `int` обычно отводится память, соответствующая "естественной" единице памяти используемой машины. Размер области памяти, выделяемой для переменных типов `short int` и `long int`, зависит от реализации языка.¹ Использование переменных типа `short int` вместо переменной `int` может как обеспечить экономию памяти, так и не привести к экономии (см. приложение Д). Однако использование переменных типа `short int` в некоторых случаях может увеличить время выполнения программы, так как в арифметических операциях значения типа `short int` сначала преобразуются в значения типа `int` до их использования. Следовательно, переменные типа `short int` должны употребляться только при необходимости экономии памяти.

Если знаковый бит в представлении числа не нужен, то переменной можно присвоить тип `unsigned int` (целый без знака), или просто `unsigned` (беззнаковый). Целые без знака используются при работе с отдельными битами или их группами в машинном слове. Они также позволяют "расширить" разрядность чисел, представимых машинным словом, за счет знакового разряда, если знак числа не нужен [76].

2.1.3. Перечисляемые типы

Перечисляемые типы позволяют в качестве значений использовать идентификаторы. Применение перечисляемых типов вносит ясность в программу, так как с их помощью значениям, смысл которых трудно определить без дополнительного пояснения, присваиваются выразительные имена. Например, идентификаторы `jan`, `feb`, `mar` для обозначения месяцев более выразительны, чем целые числа 1, 2, 3.

Множество значений, ассоциированное с перечисляемым типом, может быть описано явно перечислением значений. Описания перечисляемого типа² имеют следующий вид:

```
typedef enum {a0, a1, ..., an} E;
```

¹ $S(\text{short int}) \leq S(\text{int}) \leq S(\text{long int})$, где $S(x)$ - размер области памяти, отводимой переменной типа x .

² Механизм описания типа с помощью ключевого слова `typedef` более детально обсуждается в разд. 2.3.

где E является описываемым перечисляемым типом. Перечисляемый литерал a_i обычно представляется целым числом.¹

Ниже приведены два примера описаний перечисляемого типа:

```
typedef enum {mon, tue, wed, thu, fri, sat, sun} day;  
typedef enum {red, yellow, green} traffic_light;
```

Перечисляемый литерал не может быть ассоциирован с двумя различными типами. Например, было бы ошибкой, если бы при наличии описания типа `traffic_light` встретилось описание типа

```
typedef enum {yellow, blue, red} color;
```

поскольку `yellow` и `red` уже ассоциированы с перечисляемым типом `traffic_light`.

Переменные перечисляемых типов `day` и `traffic_light` могут быть определены как

```
day d;  
traffic_light signal;
```

Использование перечисляемого типа показано на следующем примере:

```
switch (signal) {  
case red: тормозить; подождать зеленого света; break;  
case yellow: остановиться, если возможно;  
              иначе продолжать движение; break;  
case green: движение разрешено; break;  
default: ошибка;  
}
```

Часть операторов являются абстрактными операторами. В дальнейшем такие операторы будут использоваться при необходимости пояснения работы программы или в процессе их разработки. Перед компиляцией программы абстрактные операторы

¹ Внутреннее представление перечисляемых литералов может быть указано явно. Например, в описании типа

```
typedef enum {a0=v, a1, ..., an} E;
```

перечисляемый литерал a_i представляется целым числом $v+i$.

ры должны быть заменены операторами языка Си, выполняющими те же самые действия.

Для определения переменных вместо перечисляемых типов может использоваться описание метки перечисляемого типа.¹ Например, в качестве меток перечисляемого типа могут быть описаны идентификаторы `day` и `traffic_light`:

```
enum day {mon, tue, wed, thu, fri, sat, sun};
enum traffic_light {red, yellow, green};
```

Метки перечисляемого типа аналогичны перечисляемым типам. Например, переменные `d` и `signal` могут быть определены с помощью меток перечисляемого типа, описанных выше как

```
enum day d;
enum traffic_light signal;
```

В настоящее время в языке Си перечисляемые переменные и константы рассматриваются как переменные и константы типа `int`. Следовательно, ошибочное использование перечисляемых значений в качестве целых чисел не может быть обнаружено компиляторами языка Си. Например, бессмысленное выражение

```
tue + sat
```

в котором складываются два дня недели, не будет считаться ошибочным.

Другим ограничением перечисляемых типов в языке Си является отсутствие механизма генерирования элементов перечисляемого типа. Например, невозможно написать цикл, выполняющийся один раз для каждого значения перечисляемого типа `day`; такой цикл мог бы иметь вид

```
for d in day { ... }
```

где `d` - переменная цикла, которой присваиваются различные элементы типа `d` при каждом выполнении цикла.

2.1.4. Булевы, или логические значения

В языке Си нет булевых значений, вместо них используются целые значения. Ненулевые значения соответствуют логи-

¹ При наличии более общего описания с помощью оператора `typedef` механизм описания с помощью меток перечисляемого типа является лишним. Обе возможности появились в языке Си в процессе его эволюции.

ческому значению true (истина), а ноль - логическому значению false (ложь). По соглашению, заранее определенные операторы и функции возвращают в вызывающую программу значение 1 в качестве логического значения true и 0 - в качестве false.

Для ясности при обозначении булевых значений в этой книге будут использоваться константы TRUE и FALSE, определяемые операторами

```
#define TRUE    1
#define FALSE   0
```

2.1.5. Плавающая точка

Существует два вида типов с плавающей точкой - float (обычная точность) и double (двойная точность). Например,

```
float x, y;
double eps;
```

Переменные x и y определяются как переменные типа float, а eps - как переменная типа double.

В языке Си все арифметические операции с плавающей точкой выполняются с двойной точностью. Программист должен понимать, что за повышенную точность, обеспечиваемую арифметическими операциями с двойной точностью, приходится расплачиваться увеличением времени выполнения программы по сравнению с временем, необходимым для вычислений с обычной точностью. Кроме того, нужно учитывать затраты на преобразование - все переменные типа float перед выполнением операций с плавающей точкой должны быть преобразованы к типу double, а полученный результат из представления с двойной точностью перед присваиванием значения типа float должен быть преобразован к представлению с обычной точностью.¹

2.1.6. Тип void (пустой)

С помощью типа void представляется пустое множество значений. Этот тип используется в следующих случаях:

для указания типа функций, не возвращающих значений, т.е. функций, используемых в качестве подпрограмм;

¹ В реализациях языка, для которых значения float и double эквивалентны, такие преобразования не выполняются.

для указания того, что значение выражения не будет использовано, оно вычисляется только для получения побочных эффектов (см. подразд 2.5.3).

Применение типа `void` необязательно; если выражение не используется, то его значение игнорируется. Однако указание типа `void` предотвратит появление сообщений проверочной программы `lint` и некоторых компиляторов о том, что значение выражения в программе игнорируется.

2.2. ПРОИЗВОДНЫЕ ТИПЫ

2.2.1. Массивы

Массив является сложным объектом, состоящим из объектов-компонентов (называемых элементами) одного и того же типа. Простые определения массива имеют вид:

тип данных $x[n_1] [n_2] \dots [n_k]$

где x - идентификатор, определяемый в качестве имени массива, а n_i - размерности массива. Массив x называется k -мерным массивом с элементами типа *тип данных*. Элементы i -го измерения имеют индексы от 0 до n_i-1 .

Тип элемента массива может быть одним из основных типов, типом другого массива, типом указателя (`pointer`), типом структуры (`structure`) или типом объединения (`union`). Хотя элементы массива не могут быть функциями, они могут быть указателями на функции (указатели обсуждаются в подразд. 2.2.5).

Ниже приведены некоторые примеры определений массива:

```
int page[10]; /* одномерный массив с 10 элементами, */
              /* перенумерованными с 0 до 9          */
char line[81];
float big[10][10], sales[REGION][MONTHS][ITEMS];
```

В последней строке в одном определении указаны сразу два массива: массив `big` определен как двумерный и массив `sales` - как трехмерный (`REGION`, `MONTHS` и `ITEMS` являются константами, которые должны быть предварительно описаны с помощью инструкции `#define`).

Ссылки на элемент k -мерного массива x делаются с помощью следующего обозначения:

$x[i_1] [i_2] \dots [i_k]$

где i_j - целое выражение, $0 \leq i_j \leq n_j-1$, а n_j - максимальное значение j -го индекса массива x . Например,

```

page[5]
line[i+j-1]
big[i][j]

```

Указывая только первые p индексов, можно ссылаться на p -мерный подмассив k -мерного массива ($p \leq k$), например,

```

sales[i]      /* ссылка на двумерный подмассив */
              /* массива sales                */
sales[i][j]   /* ссылка на одномерный подмассив */
              /* массива sales                */
sales[i][j][k] /* ссылка на нульмерный подмассив */
              /* массива sales, т.е. просто   */
              /* на элемент массива sales     */

```

Общая форма определений массива будет обсуждаться в разд. 2.4. Существует тесная связь между массивами и указателями; она будет рассматриваться в подразд. 2.2.4.

2.2.2. Структуры

Структура (запись в терминологии языков Паскаль и Ада) - это составной объект, в который входят компоненты любых типов, за исключением функций. В отличие от массива, который является *однородным* (*homogeneous*) объектом, структура может быть *неоднородной* (*heterogeneous*). Тип структуры указывается записью вида

```

struct {
    список описаний
}

```

В структуре должен быть указан хотя бы один компонент. Указатель типа структуры используется для определения структур. Определения структур имеют следующий вид:

тип-данных *описатели*;

где *тип-данных* указывает тип структуры для объектов, определяемых в *описателях*. В своей простейшей форме *описатели* представляют собой обычные имена переменных, массивов, указателей и функций. (Точные определения будут даны в разд. 2.4.)

Например, с помощью определения

```

struct {
    double x, y;
} a, b, c[9];

```

переменные `a` и `b` определяются как структуры, каждая из которых состоит из двух компонентов `x` и `y`. Переменная `s` определяется как массив из девяти таких структур.

Из определения

```
struct {  
    int year;  
    short int month, day;  
} date1, date2;
```

следует, что каждая из двух переменных `date1` и `date2` состоит из трех компонентов: `year`, `month` и `day`.

С типом структуры может быть ассоциировано имя, которое задается описанием типа в форме

```
typedef struct {  
    список описаний  
} имя-типа-структуры;
```

В дальнейшем эти имена могут использоваться для определения структур. (Общая форма описания `typedef` будет рассматриваться далее.)

Ниже приведен пример описания типа структуры с именем `employee`:

```
typedef struct {  
    char name[30];  
    int id;  
    dept d;  
    family f;  
} employee;
```

где слова `dept` и `family` указывают типы, а именно типы структур, предварительно определенные пользователем. Тип структуры `employee` может быть использован для определения переменных; например, определение

```
employee chairperson, president, e1, e2;
```

описывает переменные `chairperson`, `president`, `e1` и `e2` как структуры типа `employee`.

Существует и другой способ ассоциирования имени с типом структуры; этот способ основан на применении *меток структуры* (*structure tag*). Метки структуры аналогичны *меткам перечисляемого типа*. Метка структуры описывается следующим образом:

```
struct метка {
    список описаний
};
```

где *метка* является идентификатором. В приведенном ниже примере слово `student` описывается как метка структуры:

```
struct student {
    char name[25];
    int id, age;
    char sex;
};
```

Метки структуры используются для определения структур записью вида

```
struct метка список-идентификаторов;
```

Например:

```
struct student s1, s2;
```

Использование меток структуры необходимо для описания рекурсивных структур (так как одного только оператора `typedef` недостаточно). В приведенном ниже примере описания рекурсивной метки структуры

```
struct node {
    int data;
    struct node *next;
};
```

метка структуры `node` действительно является рекурсивной, так как она используется в своем собственном описании, т.е. в описании указателя `next`. (Из-за наличия знака `*` переменная `next` описана как указатель на объекты типа `node`; указатели будут рассмотрены ниже.)

Структуры не могут быть прямо рекурсивными; структура типа `S` не может содержать компонент, являющийся структурой типа `S`. Однако структура типа `S` может содержать компонент, *указывающий* на структуру типа `S`.

2.2.1. Доступ к компонентам структуры. Такой доступ осуществляется с помощью специального обозначения для *выделенного компонента*, имеющего следующий вид:

`S.c`

где *s* является именем структуры или значением структуры с компонентом *s*; *s* может быть выражением, дающим в результате значение структуры; например, *s* может быть вызовом функции со структурой в качестве ее значения. К компонентам определенной выше структуры *date1* можно обратиться, указав их обозначения:

```
date1.year  
date1.month  
date1.day
```

2.2.2.2. Поля битов. Структуры могут также использоваться для обеспечения доступа к отдельным битам слова. Использование полей битов необходимо лишь в редких случаях, например при разработке программ, в которых важно экономить память, или системных программ, прямо взаимодействующих с оборудованием; в частности, при написании драйверов устройств может понадобиться доступ к конкретным битам регистра устройства.¹

Компоненты структуры могут быть размещены в словах указанием положения и числа битов, занимаемых каждым компонентом. Такие компоненты называются полями. Хотя в языке Си допускаются поля любого типа, реализации языка ограничивают возможные типы только целыми типами; в книге [77] предлагается использовать только беззнаковые поля (*unsigned*), так как целые (*int*) могут по-разному трактоваться на разных машинах. В этой же книге указывается, что реализации языка могут отличаться некоторыми ограничениями на возможное использование полей, например во всех реализациях нельзя описывать массивы полей, нельзя применять к полям оператор *определения адреса*.

В общем случае тип *структура поля* (структура, компонентами которой являются поля) задается в следующей форме:

```
struct {  
    [unsigned идентификатор1]: длина-поля1;  
    [unsigned идентификатор2]: длина-поля2;  
    .  
    .  
    .
```

¹ В компьютерах с распределением для ввода-вывода части адресного пространства оперативной памяти (memory mapped I/O), таких как VAX-11, регистр устройства является ячейкой памяти; управление вводом-выводом осуществляется изменением содержимого этих ячеек.

```

    [unsigned идентификаторn]: длина-поляn;
}

```

где *длина-поля_i* является константой, заданной целым выражением. Эта константа указывает число битов, отведенное полю битов с именем *идентификатор_i*. Поле нулевой длины обозначает выравнивание на границу следующего слова.

Поля размещаются в памяти одно за другим без промежутков; на некоторых компьютерах (например, на PDP-11 и VAX-11) они размещаются справа налево (т.е. от младших битов к старшим), на остальных машинах - слева направо. Безымянные поля (поля, для которых указана только длина) используются для задания тех битов слова, к которым не будет доступа. Длина поля не может превышать длину слова используемой машины; если поле не умещается в оставшиеся от предыдущего поля биты слова, то оно размещается в следующем слове.

В качестве примера ниже показана структура полей `save_211`, заданных в соответствии с форматом регистра состояния диска RX211, используемого с компьютером VAX-11:

```

struct {
    unsigned error : 1; /* бит 15; в компьютере VAX-11 */
                        /* биты нумеруются справа налево */
    unsigned initialize : 1; /* бит 14 */
    unsigned unibas_addr : 2; /* биты 13-12 */
    unsigned rx02 : 1; /* бит 11 */
    unsigned : 2; /* поле не поименовано, так */
                  /* как биты 10-9 не исполь- */
                  /* зуются */
    unsigned density : 1; /* бит 08 */
    unsigned trans_reg : 1; /* бит 07 */
    unsigned int_enable : 1; /* бит 06 */
    unsigned done : 1; /* бит 05 */
    unsigned unit_select : 1; /* бит 04 */
    unsigned function : 3; /* биты 03-01 */
    unsigned go : 1; /* бит 00 */

} save_211; /* формат регистра состояния диска RX211 */

```

Структуры полей могут содержать и обычные компоненты (например, знаковые). Такие компоненты автоматически размещаются на соответствующих границах слов, при этом некоторые биты могут остаться неиспользованными.

Ссылки на поля битов выполняются точно так же, как и на компоненты обычных структур.

2.2.3. Объединения

Объединение подобно структуре, однако в каждый момент времени может использоваться (или является активным) только один из его компонентов. Тип объединения может задаваться записью вида

```
union {  
    описание компонента1;  
    описание компонента2;  
    .  
    .  
    .  
    описание компонентаn;  
}
```

Для каждого из этих компонентов выделяется одна и та же область памяти, т.е. они перекрываются. Хотя доступ к этой области памяти возможен с использованием любого из компонентов, компонент для этой цели должен выбираться так, чтобы полученный результат не был бессмысленным.

Как уже упоминалось, объединения подобны структурам. Доступ к компонентам объединения осуществляется тем же способом, что и к структурам. *Метки объединения (union tag)* могут быть описаны точно так же, как и метки структуры.

Объединения применяются для

минимизации используемого объема памяти, если в каждый момент времени только один объект из многих является активным;

интерпретации основного представления объекта одного типа, как если бы этому объекту был присвоен другой тип.

Роль объединения аналогична роли описания эквивалентности (equivalence) в Фортране. Компоненты *a* и *b* в объединении взаимосвязаны точно так же, как два объекта, связанные отношением эквивалентности в Фортране.

В качестве примера определения объекта типа `union` рассмотрим объединение `geom_fig`, определяемое следующим образом:

```
union {  
    float radius;    /* окружность */  
    float a[2];      /* прямоугольник */  
    int b[3];        /* треугольник */  
    position p;      /* точка */  
                    /* "position" - тип, */
```

```

/* описанный пользователем */
} geom_fig;

```

В этом примере не лишена смысла лишь обработка только активного компонента, т.е. компонента, который последним получил свое значение. Например, после присваивания значения компоненту `radius`, не имеет смысла обращение к массиву `b[3]`.

2.2.4. Переменные структуры

Очень часто некоторые объекты программы относятся к одному и тому же классу, отличаясь лишь некоторыми деталями. Например, рассмотрим представление геометрических фигур. Общая информация об этих фигурах может включать такие элементы, как площадь и периметр. Однако соответствующая информация об их размерах может оказаться различной в зависимости от их формы.

В языках, таких как Ада и Паскаль, имеется тип данных, называемый *переменная запись* (*variant record*), объекты которой содержат набор одних и тех же компонентов плюс компоненты, не являющиеся общими для всех остальных объектов. В языке Си также имеется тип данных, подобный переменной записи, называемый *переменной структурой* (*variant structure*), которая может быть реализована с использованием комбинации структуры и объединения. Рассмотрим для примера структуру `fig`:

```

typedef struct {
    float area, perimeter; /* общие компоненты */

    int type; /* метка активного компонента */

    union { /* переменный компонент */
        float radius; /* окружность */
        float a[2]; /* прямоугольник */
        float b[3]; /* треугольник */
        position p; /* точка */
    } geom_fig;
} figure;

```

В общем случае каждый объект типа `figure` будет состоять из трех компонентов: `area`, `perimeter` и `type`. Компонент `type` называется *меткой активного компонента* (*active component tag*), так как он используется для указания, какой из компонентов объединения `geom_fig` (например, `radius`, `a`, `b` или `p`) является активным в данный момент. Такая структура называется *переменной структурой*,

потому что ее компоненты меняются в зависимости от значения метки активного компонента.

Предположим, что даны следующие определения констант:

```
#define CIRCLE 1
#define RECT 2
#define TRIANGLE 3
#define POINT 4
```

и что переменная `fig` определяется как

```
figure fig;
```

Тогда, по соглашению, перед присваиванием значения одному из компонентов соответствующее значение присваивается также переменной `fig.type` для указания активного компонента, например,

```
fig.type = CIRCLE;
fig.geom_fig.radius = 5.0;
```

Аналогично перед обращением к компоненту объединения необходимо проверить, является ли этот компонент активным, например,

```
switch (fig.type) {
case CIRCLE: обработать окружность; break;
case RECT: обработать прямоугольник; break;
case TRIANGLE: обработать треугольник; break;
case POINT: обработать точку; break;
default: ошибка;
}
```

Для присваивания значений константам `CIRCLE`, `RECT`, `TRIANGLE` и `POINT` можно было бы использовать перечисляемый тип, например,

```
typedef enum {CIRCLE, RECT, TRIANGLE,
              POINT} figure_class;
```

Тогда компонент типа `figure` можно было бы описать как переменную типа `figure_class`. Использование перечисляемого типа `figure_class` позволит компилятору Си предупредить программиста о потенциально ошибочных присваиваниях, таких как

```
fig_type = 44;
```

В общем случае переменные структуры будут состоять из трех частей: набора общих компонентов, метки активного компонента и части с меняющимися компонентами. Общая форма переменной структуры имеет следующий вид:

```
struct {  
    общие компоненты;  
    метка активного компонента;  
    union {  
        описание компонента1;  
        описание компонента2;  
        .  
        .  
        .  
        описание компонентаn;  
    } идентификатор;  
}
```

Ниже приведен пример определения переменной структуры `health_record`:

```
struct {  
    /* общая информация */  
    char name[25];  
    int age;  
    char sex;  
  
    /* метка активного компонента */  
    marital_status ms;  
  
    /* переменная часть */  
    union {  
        /* холост */  
        /* нет компонентов */  
  
        /* женат */  
        struct {  
            char marriage_date[8];  
            char spouse_name[25];  
            int no_children;  
        } marriage_info;  
  
        /* разведен */  
        char date_divorced[8];  
    } marital_info;  
} health_record;
```

где тип `marital_status`, т.е. тип метки активного компонента `ms`, описан как

```
typedef enum {SINGLE, MARRIED, DIVORCED}
            marital_status;
```

Ниже приведены несколько примеров ссылки на компоненты переменной структуры:

```
helth_record.name
helth_record.ms
helth_record.marriage_info.marriage_date
```

2.2.5. Указатели

Указателем называется компонент заданного типа, являющийся ссылкой на некоторую область памяти. Определение указателя имеет следующий вид:

тип-данных $*id_1, *id_2 \dots, *id_n$;

Тип переменных $id_1, id_2 \dots, id_n$ определяется как тип *указателей на тип-данных* (обозначаемых как *тип-данных **); эти переменные служат ссылками на объекты типа *тип-данных*; этот тип называется *базовым* типом переменных-указателей.

Ниже приведены несколько примеров определений указателей:

```
int *pi, *qi; /* указатели на целые объекты */

struct { int x, y; } *p;
    /* указатель на структуру с */
    /* компонентами x и y */

complex *x; /* указатель на объект определенного */
            /* пользователем типа complex */
```

2.2.5.1. Динамические объекты. Указатели используются при создании и обработке динамических объектов. Заранее определяемые объекты создаются с помощью определений.¹ Динамические объекты, в отличие от заранее определяемых,

¹ Заранее определенные объекты называются также статическими объектами. Прилагательное статический (`static`) не будет использоваться во избежание возможной путаницы с классом памяти типа `static` в языке Си.

создаются динамически и явно в процессе выполнения программы. Для создания динамических объектов служат функции `malloc` и `calloc`. В отличие от заранее определенных объектов, число динамических объектов не фиксировано тем, что записано в тексте программы, - по желанию динамические объекты могут создаваться и уничтожаться в процессе ее выполнения. Динамические объекты, в отличие от заранее определенных, не имеют имен, и ссылка на них выполняется с помощью указателей.

Значение 0 может быть присвоено указателям любого типа. Это значение показывает, что данный указатель не содержит ссылки на какой-либо объект. Попытка использовать это значение для обращения к динамическому объекту приведет к ошибке. По соглашению, для обозначения константы с нулевым значением используется идентификатор `NULL`. Его описание имеет следующий вид:

```
#define NULL 0
```

и содержится среди описаний файла стандартного пакета ввода-вывода `stdio.h`.¹

2.2.5.2. Создание динамических объектов. Функции `malloc` и `calloc` имеют следующие спецификации:²

```
char *malloc(size)
unsigned size;
    /* объем памяти, который необходимо выделить */

char *calloc(nelem, elsize)
unsigned nelem;
    /* число элементов, для которых нужно выделить память */
unsigned elsize;
    /* объем памяти, который необходимо выделить */
    /* для каждого элемента */
```

Обе функции возвращают знаковый указатель, указывающий на выделенную память.

Для определения необходимого объема памяти можно использовать оператор `sizeof`.³

¹ Заметим, что значение `NULL` отлично от нулевого знака `\0`.

² По соглашению, функциональная спецификация содержит первую строку определения функции и описания параметров функции. (более подробно - см. гл. 5).

³ В первой форме оператора `sizeof` скобки не требуются; однако здесь скобки все-таки используются, чтобы обе формы оператора `sizeof` имели одинаковый вид.

| Форма | Значение |
|---------------------------------|---|
| <code>sizeof (выражение)</code> | Объем памяти, необходимый для хранения выражения |
| <code>sizeof (T)</code> | Объем памяти, необходимый для хранения значений типа <i>T</i> |

Функции `malloc` и `calloc` возвращают указатель на созданный динамический объект. Фактически функции возвращают знаковые указатели, которые могут быть явно преобразованы к подходящему типу указателя (более подробно - см. разд. 2.5.3). Значения, возвращенные функциями распределения памяти, используются для ссылок на динамические объекты. Например, с помощью оператора

```
pi = (int *) malloc(sizeof(int));
```

выделяется память для одного целого значения. Адрес этой области памяти присваивается переменной `pi` после его преобразования из типа `char *` (указатель на знак), с которым он возвращается функцией `malloc`, к типу `int *` (указатель на целое), т.е. типу переменной `pi`.

2.2.5.3. Доступ к динамическим объектам. Присваивание значения объекту, ссылка на который задана указателем `pi`, выполняется с помощью имени указателя `*pi`, например:

```
*pi = 55;
```

Одно и то же значение может быть присвоено более чем одной переменной-указателю. Таким образом, можно ссылаться на динамический объект с помощью более чем одного указателя. Про объект, к которому можно обращаться с использованием более чем одного указателя, говорят, что он имеет псевдоимена (*alias*). Например, в результате присваивания

```
qi = pi;
```

и `qi`, и `pi` указывают на один и тот же объект, т.е. они являются псевдоименами. Неуправляемое использование псевдоимен может нанести ущерб пониманию текста программы, так как возможность доступа к одному и тому же объекту и его модификация с помощью различных псевдоимен не всегда очевидны при анализе части программы.

2.2.5.4. Время жизни динамического объекта. Память, занимаемая динамическими объектами, если она необходима для других целей, должна быть освобождена явным указанием. В противном случае эта память может быть потеряна, т.е. станет

невозможным ее повторное использование. Явное освобождение выполняется использованием функции `free`, которая имеет следующую спецификацию:

```
free(ptr)
char *ptr;
```

Необходимо предпринимать меры предосторожности для избежания ошибок, связанных со ссылками на объект, память для которого уже освобождена (проблема "висящей ссылки") [39].

Если реализация языка обеспечивает "сборку мусора", то память, занимаемая объектами, к которым невозможен доступ, может быть автоматически подготовлена для повторного использования. Однако в языке Си, в отличие от языков Лисп и Снобол, такая возможность отсутствует.

2.2.5.5. *Указание на заранее определенные объекты.* Указатели могут обеспечивать ссылку на заранее определенные объекты. Адрес такого объекта может быть определен использованием оператора адресации `&` (*address of operator*). Например, рассмотрим переменные `i` и `pi`, определенные как

```
int i, *pi;
```

Присваивание

```
pi = &i;
```

позволяет ссылаться на объект с именем `i` также с помощью указателя `pi`, используя обозначение `*pi`. Имена `i` и `*pi` - псевдоимена. Оператор `&` является также стандартным средством моделирования *передачи параметров по ссылке* (см. гл. 5). Однако его употребление может привести к проблеме "висящей ссылки".

2.2.5.6. *Указание на произвольную ячейку памяти.* С помощью явных преобразований можно получить указатель на произвольную ячейку памяти. Например, предположим, что `pt` является указателем типа `T *`; тогда указатель на ячейку памяти 0777000 можно получить с помощью следующей записи:

```
pt = (T *) 0777000;
```

Обращение к конкретным ячейкам памяти часто бывает необходимо в программах, взаимодействующих с оборудованием, например в драйверах устройств, когда для управления устройствами нужно иметь доступ к таким ячейкам памяти, как регистры состояния или ячейки буфера устройства. Хотя такие возможности полезны и даже необходимы для некоторых при-

ложений, пользоваться ими следует с осторожностью. (Большинство операционных систем запрещают обычным пользователям доступ к абсолютным адресам памяти для сохранения целостности системных средств и защиты других пользователей.)

2.2.5.7. Связь между указателями и массивами. В языке Си массивы и указатели тесно связаны. Имя каждого массива может рассматриваться как указатель на первый элемент массива. Элемент массива $a[i]$ есть элемент массива, на который указывает значение $a+i$, т.е. $*(a+i)$, где значение a является адресом первого элемента массива a , а именно $a[0]$. Выражение $a+i$ является примером арифметических действий с указателями - целое значение i складывается с значением указателя, адресом первого элемента массива a . Значение этого выражения есть a плюс объем памяти, занимаемый i элементами массива a . Более детально арифметические действия с указателями рассматриваются в гл. 3.

Предположим, что x - двумерный массив. Тогда ссылка на подмассив $x[i]$ является ссылкой на i -ю строку массива x ; $x[i]$ дает адрес первого элемента этой строки, т.е. $*(x+i)$. Элементы каждой строки занимают непрерывную область памяти, так как массивы хранятся записанными по строкам, т.е. при записи элементов массива в память быстрее всех изменяется последний индекс.

Аналогично ссылка на $y[i]$, где y - n -мерный ($n > 1$) массив, является ссылкой на $(n-1)$ -мерный подмассив с элементами $y[i, j_2, j_3, \dots, j_n]$, где значения j_k соответствуют определению массива y ; $y[i]$ дает адрес первого элемента этого подмассива, т.е. $*(y+i)$. Все элементы этого $(n-1)$ -мерного подмассива занимают непрерывную область памяти.

2.2.5.8. Строки - дополнительные сведения о тесной связи между указателями и массивами. Строки - это массивы знаков. По соглашению, последним знаком строки должен быть нулевой знак $\backslash 0$. Поскольку имя массива фактически является указателем на первый элемент массива, переменные типа `string` могут также рассматриваться, как имеющие тип `char *`. Например, вторая переменная `string_array` в определении

```
char *string_pointer, string_array[81];
```

может рассматриваться также как знаковый указатель. Для строки, представленной первой переменной `string_pointer`, память должна быть выделена явно; с другой стороны, для массива `string_array` память уже выделена и переменная `string_array` является указателем на нее. Заметим, что память должна быть также выделена или зарезервирована для признака конца строки $\backslash 0$.

Нет ничего необычного не только в интерпретации переменных типа `string` (т.е. массивов знаков) как указателей, но и в интерпретации строк, которые также могут рассматриваться двояко - как массивы и как указатели - и все в одной программе! Это особенно важно, когда строки передаются как аргументы функции. Вызывающая программа может рассматривать строку как массив знаков, а вызываемая функция может рассматривать ее как знаковый указатель.

Если длина строки непостоянна, то использование знаковых указателей для строк имеет определенные преимущества. Хотя строки переменной длины могут быть также реализованы с использованием массивов, такая реализация оказывается слишком неэкономной с точки зрения использования памяти и налагает ограничения на максимальную длину строки. Например, для размещения строк разной длины может быть создан массив знаковых указателей; альтернативное решение с использованием двумерного массива знаков в общем случае будет использовать память неэффективно, так как в этом случае потребовалось бы сделать число столбцов равным числу знаков в строке наибольшей возможной длины.

2.2.5.9. Указатели и структуры. Рассмотрим метку структуры `student`, описание которой было дано выше как

```
struct student {  
    char name[25];  
    int id, age;  
    char sex;  
};
```

и указатель `new_student`, определенный как

```
struct student *new_student;
```

Предположим, что память выделена таким образом, чтобы `new_student` указывал на объект `student`. Тогда на компоненты этого объекта можно ссылаться следующим образом:

```
(*new_student).name  
(*new_student).id  
(*new_student).age  
(*new_student).sex
```

Поскольку указатели часто используются для указания на структуры, в язык Си специально для ссылок на компоненты таких структур введен оператор выбора *стрелка вправо*. Например, ссылки на вышеприведенные компоненты структуры

можно записать с использованием оператора *стрелка вправо* -> как

```
new_student->name
new_student->id
new_student->age
new_student->sex
```

В качестве примера, иллюстрирующего связь между структурами и указателями, рассмотрим описание типа `stat_reg`:

```
typedef struct {

    unsigned error: 1; /* бит 15; на компьютере VAX-11 */
                        /* биты нумеруются справа налево */
    unsigned initialize: 1; /* бит 14 */
    unsigned unibus_addr: 2; /* биты 13-12 */
    unsigned rx02: 1; /* бит 11 */
    unsigned: 2; /* поле не поименовано, */
                /* так как биты 10-9 */
                /* не используются */
    unsigned density: 1; /* бит 8 */
    unsigned trans_reg: 1; /* бит 7 */
    unsigned int_enable: 1; /* бит 6 */
    unsigned done: 1; /* бит 5 */
    unsigned unit_select: 1; /* бит 4 */
    unsigned functin: 3; /* биты 3-1 */
    unsigned go: 1; /* бит 0 */

} stat_reg; /* формат регистра состояния диска RX211 */
```

Тип `stat_reg` используется для определения указателя `ptr_rx_sr`:

```
stat_reg *ptr_rx_sr;
```

Указатель `ptr_rx_sr` введен для ссылки на регистр состояния диска RX211, находящегося по адресу 0777170, с помощью присваивания

```
ptr_rx_sr = (stat_reg *) 0777170;
```

Теперь к компонентам регистра можно обращаться следующим образом:

```
ptr_rx_sr->error
ptr_rx_sr->density
ptr_rx_sr->int_enable
```

и так далее.

2.3. ОПИСАНИЯ ТИПА

Описания типа используются для того, чтобы все свойства объекта собрать в одном месте и присвоить им имя. Это *имя типа* может затем использоваться для последующих описаний других объектов. Описания типа имеют следующую форму:

```
typedef спецификатор-типа описатели;
```

где *спецификатор-типа* - основной или производный тип, или тип, ранее определенный программистом. Описываемые имена типов представляются идентификаторами в описателях.

Ниже приведены несколько описаний типов:

```
typedef float miles, speed;
/* имена "miles" и "speed" описаны как */
/* синонимы для имени "float" */
```

```
typedef float a[5], *pf;
/* тип "a" описан как массив из 5 ком- */
/* понентов типа float; тип "pf" описан */
/* как указатель на объекты типа float */
```

```
typedef struct { float x, y; } point;

/* тип структуры "emp" описывает размещение за- */
/* писей в файле базы данных; используемые здесь */
/* константы были определены ранее */
```

```
typedef struct {
    char name[LN];
    char room[LR];
    char ext[LE]; /* расширение */
    char desig[LD]; /* обозначение */
    char compid[LC]; /* идентификатор компании */
    char sig[LS]; /* электронная подпись */
    char logid[LL]; /* идентификатор для входа */
    /* в систему */
    char maild[LM]; /* каталог, куда принимает- */
    /* ся почта */
} emp;
```

С помощью этих типов можно определять объекты точно так же, как с помощью типов float и int. Например, оператором

```
point s1, s2, *p;
```

s1 и s2 определяются как структуры типа point, а p - как указатель на структуру типа point.

Описаниями типа в языке Си новые типы не вводятся; вместо них задаются синонимы для уже существующих типов или вводятся типы, которые могут быть образованы из существующих типов. Например, описанные выше типы miles и speed являются синонимами как друг для друга, так и для заранее определенного типа float; все они эквивалентны.

2.4. ОПРЕДЕЛЕНИЯ И ОПИСАНИЯ

Переменная состоит из двух компонентов: объекта и имени этого объекта. Имена могут быть идентификаторами или выражениями. Например, выражение

```
*p
```

является именем объекта, ссылка на который осуществляется с помощью указателя p.

Идентификаторы определяются и описываются с помощью определений и описаний в следующей форме:

класс-памяти тип-данных описатели;

где *класс-памяти* и *тип-данных* могут быть опущены. Если они заданы, то относятся к каждому описателю, характеризующему, в свою очередь, идентификатор. За каждым описателем может следовать *инициализатор*, задающий начальное значение, ассоциированное с идентификатором в этом описателе.

2.4.1. Классы памяти

Время жизни и область действия идентификатора определяются ассоциированным с ним *классом памяти*. Существуют четыре разновидности классов памяти:

auto

Автоматический - локальные идентификаторы, память для которых выделяется при входе в блок (т.е. составной оператор) и освобождается при выходе из блока (слово auto является сокращением слова automatic).

| | |
|----------|---|
| static | Статический - локальные идентификаторы, существующие в процессе всех выполнений блока. В отличие от идентификаторов типа auto, для идентификаторов типа static память выделяется только один раз - в начале выполнения программы, и они существуют, пока программа выполняется. |
| extern | Внешний - идентификаторы, называемые внешними (external), используются для связи между функциями, в том числе независимо скомпилированными функциями (которые могут находиться в различных файлах). Память, ассоциированная с этими идентификаторами, является постоянной, однако ее содержание может меняться. Эти идентификаторы описываются вне функций. |
| register | Регистровый - идентификаторы, подобные идентификаторам типа auto; их значения, если это возможно, должны помещаться в регистрах машины для обеспечения быстрого доступа к данным. |

Если класс памяти идентификатора не указан явно, то его класс памяти задается положением его определения в тексте программы; если идентификатор определяется внутри функции, тогда его класс памяти auto, в остальных случаях идентификатор имеет класс памяти extern.

Предположим, что имеется программа на языке Си, исходный текст которой содержится в нескольких файлах. Для разделения данных (для связи) в функциях в этих файлах используются идентификаторы, определенные как extern. Если функция ссылается на внешний идентификатор, то файл, содержащий его, должен иметь описание или определение этого идентификатора. Явное задание класса памяти extern указывает на то, что этот идентификатор определен в другом файле и здесь ему память не выделяется, а его описание дано лишь для проверки типа и для генерации кода.¹ Для внешнего идентификатора память выделяется только в том случае, если класс памяти не

¹ При описании внешних массивов максимальное значение первого индекса массива указывать нет необходимости, оно будет получено из соответствующего определения, что создает дополнительные удобства для пользователя.

указан явно.¹ Хотя описание внешнего идентификатора может встретиться во многих файлах, только один файл должен содержать определение внешнего идентификатора.

Как следует из вышеприведенного обсуждения, область действия внешних идентификаторов не ограничивается файлом, содержащим их определения, а включает также файлы с соответствующими описаниями (с классом памяти `extern`). Однако определение внешних переменных как статических ограничивает их область действия файлом, содержащим эти переменные.

В качестве примера рассмотрим описания и определения переменных `x` и `y` вне функций в двух файлах `a.c` и `b.c`. Файл `a.c` содержит определения

```
int x;  
static int i;  
.  
.  
.
```

а файл `b.c` содержит описание и определение

```
extern int x;  
static int i;  
.  
.  
.
```

Переменная `x` разделяется двумя этими файлами; память для нее выделяется в определении в файле `a.c`, и эта переменная может использоваться для связи между функциями в этих двух файлах. С другой стороны, каждый файл имеет собственную локальную переменную `i`; память для нее выделяется в каждом из двух определений переменной `i`. Каждая переменная `i` доступна только функциям из файла, содержащего ее определение.

Поскольку в большинстве компьютеров доступны лишь некоторые регистры, в них могут храниться только несколько переменных. Описание переменной как регистровой может способствовать увеличению скорости выполнения программы только в случае использования компилятора языка Си, не

¹ Явное указание класса памяти `extern` является отличительным признаком внешнего описания от внешнего определения.

оптимизирующего программу, или с незначительной оптимизацией. В некоторых случаях описание переменных как регистровых может фактически замедлить выполнение программы.¹ На использование регистровых переменных накладываются определенные ограничения; например, невозможно определение адреса переменной, размещенной в регистре (с помощью оператора адресации &); более того, в регистрах могут храниться только переменные простых типов, таких как `int` и `char`, а также значения указателей.

2.4.2. Типы данных

Тип данных, указанный в определении или описании, может быть одним из следующих:

```
char  
int  
short int (или просто short)  
long int (или просто long)  
unsigned int (или просто unsigned)  
float  
double (или long float)  
void  
struct метка  
union метка  
enum метка  
typedef-имя
```

где *метки* и *typedef-имя* должны быть предварительно описаны.

По умолчанию (если в определении или описании тип данных не указан) предполагается тип `int`.

2.4.3. Описатели

Каждый описатель содержит только один идентификатор, который является именем объекта, определяемого или описываемого этим описателем. Описатели должны разделяться запятыми:

описатель, описатель, ..., описатель

¹ Предположим, что для малопользуемой переменной программист задает класс памяти `register`. Выделение регистра для такой переменной замедлит выполнение программы, если это будет мешать лучшему использованию регистров. Заметим, что для некоторых компиляторов описания переменных как регистровых не нужны.

Форма и семантика описателей объясняются ниже. Предположим, что *T* - тип данных, заданный определением объекта:

| | |
|--|--|
| <i>идентификатор</i> (<i>описатель</i>) | определяется <i>идентификатор</i> типа <i>T</i> то же самое, что и <i>описатель</i> |
| * <i>описатель</i> | то же самое, что и <i>описатель</i> в определении объекта с типом данных <i>указатель на T</i> |
| <i>описатель</i> () | то же самое, что и <i>описатель</i> в определении объекта с типом данных <i>функция, возвращающая значение типа T</i> |
| <i>описатель</i> [<i>N</i>] | то же самое, что и <i>описатель</i> в определении объекта с типом данных <i>массив с N элементами типа T</i> ; <i>N</i> - выражение с постоянным значением, элементы нумеруются от 0 до <i>N</i> -1 |

Описатели используются с некоторыми ограничениями. Функции не могут возвращать массивы или функции как значения (однако допустимы указатели на массивы или указатели на функции). Кроме того, не могут быть описаны или определены массивы функций, не могут быть функции и компонентами структуры или объединения.

При использовании в описаниях и определениях оператор косвенной ссылки *** имеет меньший приоритет по сравнению с двумя другими операторами, а именно "[]" (для обозначения массива) и "()" (для обозначения функций).

2.4.4. Примеры определений и описаний объектов

Определение¹

```
int i, *ip, f(), *fip(), (*pfi)();
```

содержащее описатели *i*, **ip*, *f()*, **fip()*, и *(*pfi)()*, приводит к следующим определениям и описаниям² идентификаторов в этих описателях:

¹ Семантика определений и описаний зависит от того, где они находятся - внутри функции или вне ее. Если из контекста или с учетом явного указания не будет ясно, что определения и описания находятся вне функции, то будет предполагаться, что они находятся внутри функции.

² Определения и описания могут быть перемешаны, что и иллюстрируется ниже-приведенным гибридом определения-описания:

```
int i, *ip, f(),(*pfi)();
```

В общем случае я буду называть такой гибрид определением. В этом определении идентификаторы *i*, *ip* и *pfi* определяются, в то время как идентификаторы *f* и *fip* опи-

| | |
|-----|---|
| i | целая переменная |
| ip | указатель на целую переменную |
| f | функция, возвращающая целое значение |
| fip | функция, возвращающая указатель на целое значение |
| pfi | указатель на функцию, возвращающую целое значение |

По умолчанию, тип всех этих переменных определен как auto.
В определении

```
static emp *db[MAX_DB];
```

db определен как массив указателей на элементы типа emp.
Элементы массива db нумеруются от 0 до MAX_DB-1.
Оператор

```
static int size, cur = 0;
```

определяет size и cur как статические переменные типа int.
Переменной cur присваивается начальное значение, равное 0.

2.4.4.1. Синтаксическое отличие описаний объекта от его определений. По некоторым признакам, приведенным ниже, можно заметить синтаксические отличия описания объекта от его определения:

1. Наличие ключевого слова extern указывает на то, что объекты описываются, а не определяются.

2. Отсутствие параметров функции и соответствующего имени тела функции означает, что функция описывается, но не определяется.

3. Параметры функции описываются, но не определяются.

Ниже приведены несколько примеров описаний и определений:

```
/* описания */
```

```
char *strcat(), *index(), *strcpy();
extern int max(), no_of_processes;
extern float a[];
```

```
/* определения */
```

```
struct node *head;      /* определение структуры с */
                        /* использованием метки "node" */
```

сываются. Идентификаторы f и fip ссылаются на функцию, для которой память здесь не выделяется, эти функции определяются где-нибудь в другом месте.

```

point x, y;           /* определение структуры с      */
                      /* использованием типа "point" */

static union {
    automobile a;
    bus b;
    truck t;
} vehicle;

day d;
emp list_of_emp[100];

```

2.4.5. Инициализаторы

Для присваивания начальных значений переменным при их определении используются инициализаторы. Они имеют следующую форму:

= *значение*

= { *список значений* }

Значения друг от друга отделены запятыми, а каждое из них представляет собой выражение с постоянным значением. Значение может быть *сложным*, т.е. представлять собой список значений, заключенный в фигурные скобки. Статическим и внешним переменным по умолчанию присваиваются нулевые значения, в то время как автоматическим переменным никаких значений по умолчанию не присваивается.¹ Более того, нельзя задать начальные сложные значения типа auto.

Ниже приведены несколько примеров присваивания переменным начальных значений:

```

static float eps = 0.0001;

int i = 0, j = 0;

int year[12] = {31, 28, 31, 30, 31, 30, 31,
               31, 30, 31, 30, 31};

char greetings[21] = "Welcome to Bell Labs";
/* длина строки равна 21 - один элемент */
/* резервируется для признака конца строки */

```

¹ Для внесения большей ясности в программу начальные значения переменным до их использования всегда будут присваиваться явно.

```

char error[] = "Buffer length exceeded";
/* указывать размерность массива нет */
/* необходимости, так как компилятор */
/* получит его из начальных значений */

/* пример вложенной инициализации */

float matrix[5][3] = { {1.0, 1.0, 1.0},
                       {2.0, 2.0, 2.0},
                       {3.0, 3.0, 3.0},
                       {4.0, 4.0, 4.0},
                       {5.0, 5.0, 5.0}
                     };
/* элементам строки i присваивается */
/* значение i */

point p = {0.0, 0.0};
/* структура типа "point" была */
/* описана ранее */

```

2.4.6. Комментарии к синтаксису описаний и определений

В языке Си синтаксис описаний и определений объектов аналогичен синтаксису, используемому для доступа к значениям этих объектов, т.е. аналогичен синтаксису представления объектов в выражениях. В этом состоит важное отличие между языком Си и такими языками, как Алгол 68, Паскаль и Ада, в которых синтаксис, используемый для описания типов объектов, отражает структуру типа. Описания и определения в языке Си иногда трудно читать и понимать, особенно если они включают выражения составных типов, содержащих оператор косвенной ссылки *. Основная причина возникновения этих трудностей состоит в том, что оператор косвенной ссылки является префиксным оператором, в то время как все остальные операторы являются постфиксными. Читателю, не имеющему достаточно опыта программирования на языке Си, поначалу могут показаться трудными для понимания такие определения и описания, как

```

int (*(x)[6])();
char (*(y())[3])(); /* цитируется по [2] */

```

где

1. Переменная *x* определяется как указатель на массив из 6 элементов, каждый из которых является указателем на функцию, возвращающую указатель на целый объект.

2. Переменная *y* описывается как функция, которая возвращает указатель на массив указателей на функции, возвращающие знаковые значения.

Возможно, читателю будет легче понимать описания и определения, если он заметит, что описание объекта или его определение очень похоже на запись ссылки на этот объект в выражении.

Чтобы помочь в понимании и написании определений и описаний языка Си, Андерсон [2] предлагает следующие приближительные аналоги описаний и определений языков Алгол и Паскаль (промежуточные формы описаний и определений используются с целью помочь установлению эквивалентностей):

| Синтаксис языка Си | Промежуточная форма | Аналог синтаксиса Паскаля |
|-----------------------|------------------------|---|
| <code>int x</code> | | <code>x: целое</code> |
| <code>int *x</code> | <code>*x: int</code> | <code>x: указатель на целое</code> |
| <code>int x[]</code> | <code>x[]: int</code> | <code>x: массив[] целых</code> |
| <code>int x()</code> | <code>x(): int</code> | <code>x: функция(), возвращающая целое</code> |

где знак ":" должен интерпретироваться как "типа".

Тип `int` используется только для иллюстрации. Аналогичные эквивалентности можно привести и для других типов.

Преобразуя описания и определения, подобные имеющимся в языке Паскаль, в описания и определения языка Си и обратно, помните, что оператор косвенной ссылки `*` имеет приоритет меньше, чем операторы `[]` и `()`, используемые в описаниях или определениях соответственно массивов и функций. Например, в процессе грамматического разбора описание

```
char *a[];
```

воспринимается как

```
char *(a[]);
```

которое описывает *a* как массив указателей на знаки, а не как указатель на массив знаков:

```
char (*a)[];
```

Для явного указания приоритета операторов или для большей ясности скобки могут использоваться без ограничений.

2.4.7. Примеры, иллюстрирующие использование эквивалентностей

Определения и описания в языке Си можно легко создавать и понимать. Чтобы это показать, воспользуемся эквивалентностями, предложенными Андерсоном.

Определение

```
int *(*(*x)[6])();
```

приведенное выше, легко воспринимается с помощью конструирования эквивалентного определения, аналогичного определению языка Паскаль (ниже показаны последовательные преобразования исходного определения):

```
 *(*(*x)[6])(): int;  
(*(*x)[6])(): указатель на целое;  
(*(*x)[6]): функция, возвращающая указатель на целое;  
 *(*x)[6]: функция, возвращающая указатель на целое;  
(*x)[6]: указатель на функцию, возвращающую указатель  
           на целое;  
(*x): массив [0..5] указателя на функцию, возвращающую  
           указатель на функцию;  
 *x: массив [0..5] указателя на функцию, возвращающую  
           указатель на функцию;  
 x: указатель на массив [0..5] указателя на функцию,  
    возвращающую указатель на целое;
```

В следующем примере у описывается как функция, которая возвращает указатель на массив указателей на функции, возвращающие знаковые значения. Описания для языка Си в этом примере будут построены с использованием вышеприведенных эквивалентностей:

```
 y: функция, возвращающая указатель на массив указателя  
           на функцию, возвращающую знак;  
 y(): указатель на массив указателя на функцию,  
           возвращающую знак;  
 *y(): массив указателя на функцию, возвращающую знак;  
 (*y()): массив указателя на функцию, возвращающую знак;  
 (*y())[ ]: указатель на функцию, возвращающую знак;  
 *(*y())[ ]: функция, возвращающая знак;  
 (*(y())[ ]): функция, возвращающая знак;
```



```
(*(*y())[ ]()): знак;  
char (*(*y())[ ]());
```

2.4.8. Использование оператора *typedef* для упрощения понимания описаний и определений

Занн [94] предложил для упрощения сложных описаний типа использовать последовательность описаний типа. Например, тип переменной *x*, определенной как

```
x: указатель на массив[0..5] указателя на функцию,  
    возвращающую указатель на целое
```

можно определить как тип *PA6PFPI*, который, в свою очередь, постепенно строится с использованием последовательности описаний типа:

```
typedef int *PI      /* PI - указатель на целое */  
  
typedef PI FPI();  
    /* FPI: функция, возвращающая указатель */  
    /* на целое                               */  
  
typedef FPI *PFPI();  
    /* PFPI: указатель на функцию,          */  
    /* возвращающую указатель на целое      */  
  
typedef PFPI A6PFPI[6];  
    /* A6PFPI: массив указателя на функцию, */  
    /* возвращающую указатель на целое      */  
  
typedef A6PFPI *PA6PFPI;  
    /* PA6PFPI: указатель на массив указателя */  
    /* на функцию, возвращающую указатель на */  
    /* целое                                   */
```

Теперь переменная *x* может быть определена как

```
PA6PFPI x;
```

Хотя применение оператора *typedef* решает до некоторой степени проблему, связанную с пониманием запутанных описаний языка Си, их использование далеко не всегда дает ясные и краткие описания.

2.4.9. Заключительные замечания об описаниях и определениях

К счастью, сложные описания в реальных программах встречаются нечасто. Большинство программистов редко будут сталкиваться с такими сложными описаниями, какие были приведены в примерах.

Теперь несколько слов о различиях в описаниях и определениях языка Си и подобных тем, что имеются в языке Паскаль. В последних все объекты имеют один и тот же тип, например, в определении

```
a, b: массив[0..5] целых;
```

как `a`, так и `b` определяются как целые массивы, и эта определяющая информация относится ко всем определяемым переменным (в Паскале такое "вынесение" допускается для определяющей информации любого типа). С другой стороны, в языке Си такой способ определения допускается только с некоторыми видами определяющей информации. Например, в определении

```
int a[6], b[6];
```

только элемент `int` может быть вынесен из определения; при указании размерности массива это невозможно, даже если оба массива имеют одинаковую размерность. (Если желательно всю общую определяющую информацию выносить из определения, то соответствующий тип должен быть описан с помощью оператора `typedef` и использоваться затем для определения объектов.)

Однако в языке Си допускается совместное описание и определение переменных различных типов, например, в определении

```
int a[6], *c;
```

`a` определяется как массив типа `int`, в то же время переменная `c` определяется как указатель на объект типа `int`. С использованием определений, подобных определениям языка Паскаль, написать такие определения невозможно.

2.4.10. Эквивалентность типов

Существует несколько схем для определения, являются ли типы двух объектов эквивалентными. Две схемы, наиболее часто используемые, называются *структурная эквивалентность типов* и *именная эквивалентность типов*. В соответствии со схемой

структурной эквивалентности типов два объекта относятся к одному и тому же типу только в том случае, если их компоненты имеют одинаковые типы; в соответствии со схемой именной эквивалентности типов два объекта имеют один и тот же тип только в случае их определения с использованием имени того же типа.

Большинство реализаций языка Си используют схему структурной эквивалентности типов. Однако в книге [77] вопрос об эквивалентности типов игнорируется, и при каждой реализации может быть выбрана своя схема определения эквивалентности типов. Следовательно, вполне возможно, что результаты правильно работающей программы станут неверными при замене компилятора.

2.5. ПРЕОБРАЗОВАНИЯ ТИПОВ

Значения могут быть преобразованы из одного типа в другой. Такое преобразование может быть неявным или явно выполнено программистом. В этой книге везде, где необходимо для ясности программы, преобразования типа будут выполняться явно даже в тех местах программы, где явное преобразование необязательно, поскольку неявное преобразование приводит к тому же результату.

2.5.1. Неявное преобразование типа

Неявные преобразования типа выполняются главным образом для согласования аргументов оператора или функции (если это возможно) с значениями, предполагаемыми в этих операторах или функциях. Все неявные преобразования типа, которые могут встретиться в этой книге, перечислены ниже (слева указывается преобразуемый тип, а справа - список типов, в которые он может быть преобразован), другие возможные неявные преобразования в этой книге употребляться не будут ¹:

| | |
|-------------------|---|
| <code>char</code> | <code>int</code> , <code>short int</code> , <code>long int</code> (преобразование к значению с большим числом двоичных разрядов может включать, а может не включать расширение знакового разряда - это зависит от реализации языка; для элементов заданного набора знаков гарантируется преобразование в неотрицательные целые значения). |
|-------------------|---|

¹ Несколько допустимых неявных преобразований здесь не указаны, поскольку их использование делает программу менее ясной и затруднит поиск ошибок. Например, опущены неявные преобразования указателей; такие преобразования должны выполняться явно.

| | |
|-----------|---|
| int | char, short int, long int (преобразование к целому большей длины включает расширение знакового разряда; преобразование к целому меньшей длины вызывает отбрасывание лишних старших разрядов); float, double; unsigned int (интерпретация комбинации битов в виде беззнакового целого значения). |
| short int | аналогично типу int. |
| long int | аналогично типу int. |
| float | double; int, short int, long int (машинно-зависимое преобразование; если преобразуемое значение слишком велико, то результат неопределен). |
| double | float (преобразование с округлением и последующим отбрасыванием лишних разрядов); int, short int, long int (см. float). |

2.5.2. Арифметические преобразования

Арифметические операторы языка Си преобразуют операнды к соответствующим типам автоматически, если операнды не имели таких типов с самого начала. Схема преобразования, используемая этими операторами, называется *обычные арифметические преобразования*; эта схема может быть описана следующими правилами:

1. Преобразовать операнды типов char и short int к типу int; преобразовать операнды типа float к типу double.

2. Если хотя бы один из операндов имеет тип double, то и другой операнд преобразуется к типу double (если он другого типа); результат имеет тип double.

3. Если хотя бы один операнд имеет тип long, то и другой операнд преобразуется к типу long (если он другого типа); результат имеет тип long.

4. Если хотя бы один из операндов имеет тип insigned, то и другой операнд преобразуется к типу unsigned (если его тип не unsigned); результат имеет тип unsigned.

5. И, наконец, если ни один из случаев 1-4 не имеет места, то оба операнда должны иметь тип int, такой же тип будет и у результата.

2.5.3. Явные преобразования типов

Выражения могут быть преобразованы из одного типа в другой явным указанием. Выражение *E* может быть явно преобразовано к типу *иля-типа* с помощью записи вида

(иля-типа) E

где *иля-типа* представляется в форме

указатель-типа абстрактный-описатель

Абстрактный описатель аналогичен описателю, за исключением того, что он не содержит определяемого или описываемого идентификатора. Смысл слов *иля-типа*, представляемого в форме

T абстрактный-описатель

где *T* является указателем типа, может быть определен из следующей таблицы:

*Форма абстрактного
описателя*

*Пустой
(абстрактный-описатель)*

**(абстрактный-описатель)
Абстрактный описатель()*

Абстрактный-описатель[n]

Смысл слов
"Т абстрактный
описатель"
Тип *T*
Абстрактный
-описатель типа *T*
Указатель на тип *T*
Функция,
возвращающая
значение
типа *T*
Массив с *n*
элементами типа *T*,
n - выражение с
постоянным
значением

Ниже приведены несколько примеров имен типов:

```
char  
char[8]  
char *  
char()  
char *()  
char (*)()
```

Предположим, что даны следующие определения и описания:

```
int i;  
char *pc, *name;  
char *calloc(), *strcpy();
```

тогда можно привести следующие примеры явных преобразований типов:

```
(char) i /* преобразует значение типа int в значение */  
        /* типа char */
```

```
pc = (char *) 0777  
      /* преобразует восьмеричный литерал 0777 */  
      /* в значение указателя на знак таким обра- */  
      /* зом, что оно может быть присвоено */  
      /* переменной "pc" */
```

```
(emp *) calloc(1, sizeof(emp))  
      /* преобразует значение "знакового" указа- */  
      /* теля, возвращаемого функцией "calloc", */  
      /* в значение указателя "emp" */
```

```
(void) strcpy(name, "gehani");  
      /* опускает значение, возвращенное функцией */  
      /* "strcpy" */
```

Все типы могут быть явно преобразованы в тип void; однако тип void нельзя преобразовать в какой-либо другой тип. Такие преобразования для структур и объединений не реализованы [77].

2.6. ЗАДАЧИ

1. Почему в языке Си не допускается, чтобы один и тот же перечисляемый литерал входил в два различных перечисляемых типа?

2. Напишите тип переменной структуры, с помощью которой можно организовать хранение данных о различных видах транспорта: грузовиках, автобусах, легковых автомобилях и мотоциклах. Для каждого вида транспорта имеются как общие характеристики (владелец, производство и модель), так и индивидуальные (для грузовиков - число осей, грузоподъемность, для автобусов - число мест для пассажиров, для легковых автомобилей - число дверей (2 или 4), для мотоциклов - тип двигателя (двух- или четырехтактный)).

3. Исследуйте, каким образом использование оператора адресации & может привести к проблеме "висящей ссылки".

4. Рассмотрим программу:

```
#include <stdio.h>
main ()
{
    char *test[5];
    int i;

    for (i=0; i<=4; i++)
        test[i] = "0123456789";
    test[1][3]='*';
    for (i=0, i<=4; i++)
        printf("%s\n", test[i]);
}
```

Что будет получено в результате работы программы?¹ Будет ли оператор присваивания

```
test[1][3]='*';
```

оказывать действие на строку, указанную значением test[1], или на все элементы массива test? Почему? Выполните программу, воспользовавшись вашим компилятором, и проверьте результат.

Какими должны быть элементы массива test, чтобы указывать на различные строки с тем же значением?

5. Определите тип переменной k в следующем описании:

```
int *(*k())()[]
```

6. Почему приведенное ниже описание функции неправильно?

```
int ((*f()) ())[];
```

¹ Оператор цикла *for*

```
for (i=0; i<=4; i++)
    S
```

вызывает выполнение оператора S пять раз. При первом выполнении оператора S переменная i имеет значение 0; это значение увеличивается на 1 при каждом успешном выполнении оператора S.

7. Определите переменную *x* как массив указателей, возвращающих указатели на массив указателей на целые значения. При конструировании определения воспользуйтесь обозначениями, аналогичными обозначениям в языке Паскаль.

8. Для определения переменной *x* в задаче 7 воспользуйтесь последовательностью операторов `typedef`.

9. В арифметических операторах языка Си до начала вычислений операнды типа `float` преобразуются в операнды типа `double`. Каковы последствия этого автоматического преобразования? (Проанализируйте это с точки зрения точности и скорости выполнения программы.)

Глава 3

Операторы и выражения

3.1. ОПЕРАТОРЫ

Язык Си отличается большим разнообразием операторов. В этом разделе будет обсуждаться семантика этих операторов. Каждый оператор характеризуется уровнем *приоритета* и *порядком*, в котором эти операторы выполняются - слева направо или справа налево. Если все операторы выражения имеют один и тот же уровень приоритета, то значение выражения вычисляется слева направо в соответствии с порядком выполнения операторов (все операторы с одним и тем же уровнем приоритета имеют один и тот же порядок). Однако если в выражении имеются операторы с различными уровнями приоритета, то сначала выполняются операторы с наивысшим уровнем приоритета, затем - следующего за ним приоритета и так далее в порядке убывания приоритета. Операторы одного уровня приоритета выполняются в последовательности, указанной их порядком.

Ниже будут представлены группы операторов, упорядоченные в соответствии с убыванием их приоритета (вместе сгруппированы операторы с одинаковым приоритетом). Таблица приоритетов операторов будет приведена в разд 3.1.16.

3.1.1. Операторы вызова функции, индексирования и выбора

В качестве операторов языка Си рассматриваются также скобки в вызове функции, квадратные скобки для индексирования массивов, точка и стрелка вправо для выбора компонентов структуры или объединения. Эти операторы перечислены в нижеприведенной таблице, уровень их приоритета равен 1, все операторы выполняются слева направо.

| Оператор | Название | Комментарии |
|----------|--|---|
| () | Вызов функции | Например, sqrt(x), puts(s), printf("результат =%f\n", result) |
| [] | Индекс массива | Например, sales[i],sales[i][j] |
| . | Выбор компонентов структуры | Например, fig.type, (*new_student).name |
| -> | Выбор компонентов структуры (с указателем) | Например, new_student->name, где new_student - указатель на структуру |

3.1.2. Унарные операторы

Для унарных операторов требуется только один операнд; эти операторы либо префиксные, либо префиксные и постфиксные. Оператор sizeof имеет два варианта: префиксный оператор и унарная функция. В приведенной ниже таблице перечислены все эти операторы; уровень их приоритета равен 2, порядок выполнения операторов справа налево, для постфиксных операторов в таблице имеется специальное указание, остальные операторы - префиксные.

| Оператор | Название | Тип операнда | Тип результата | Комментарии |
|----------|------------------|--------------------------------------|----------------|-----------------|
| * | Косвенная ссылка | Указатель на любой тип T, кроме void | T | |
| & | Получение адреса | Переменная любого типа T, кроме void | Указатель на T | |
| - | Отри- | Арифме- | int, | Выполнены обыч- |

| | | | | |
|-------------------------------|--------------------------------|--|---|---|
| | цание | тический | unsigned, long, double int | ные арифмети- ческие преобразо- вания Если операнд равен 0, то результат ра- вен 1 и наоборот |
| ! | Логичес- кое отри- цание | Арифме- тический или указатель | | |
| ~ | Дополне- ние до 1 | Интегральный | int, long, unsigned | Выполнены обыч- ные арифметиче- ские преобразова- ния |
| ++ | Увеличе- ние | Арифме- тический или указатель | int, unsigned, long, double, указа- тель | Выполнены обыч- ные арифмети- ческие преобразо- вания; значение операнда увеличи- вается и становит- ся новым значени- ем операнда. Значе- ние указателя уве- личивается на ве- личину указывае- мого объекта; зна- чения других опе- рандов увеличива- ются на единицу |
| ++ (пост- фикс- ный) | Увеличе- ние | Арифме- тический или указатель | int, unsigned, long, double, указа- тель | Выполнены обыч- ные арифметиче- ские преобразова- ния; значение опе- ранда увеличивает- ся, но возвращает- ся старое значение операнда. Значение указателя увеличи- вается на вели- чину указываемого объекта; другие операнды увеличи- ваются на единицу |
| -- | Умень- шение | Те же, что и для ++ | Те же, что и для ++ | То же, что и для ++, но операнд уменьшается |
| -- (пост- фикс- ный) | Умень- шение | Те же, что и для ++ (пост- фикс- | Те же, что и для ++ (пост- фикс- | То же, что и для ++ (постфиксного), но операнд уменьшается |

| | | | | |
|--------|---------------------------------------|--|-------------------|---|
| sizeof | Определение требуемой памяти в байтах | ного) Значение любого типа или имени типа | ного) unsigned | Используется как sizeof(<i>выражение</i>) или sizeof(<i>иля-типа</i>) |
|--------|---------------------------------------|--|-------------------|---|

3.1.3. Мультипликативные операторы

При необходимости производятся обычные арифметические преобразования. Уровень приоритета мультипликативных операторов равен 3, порядок их выполнения - слева направо.

| Оператор | Название | Тип операндов | Тип результата | Комментарии |
|----------|-------------------|----------------|-----------------------------|---|
| * | Умножение | Арифметический | int, unsigned, long, double | |
| / | Деление | Арифметический | int, unsigned, long, double | При делении положительных чисел дробная часть отбрасывается; если хотя бы один операнд отрицателен, способ отбрасывания дробной части машинно-зависим |
| % | Получение остатка | Интегральный | int unsigned long | Знак остатка машинно-зависим |

Целочисленное деление и операции получения остатка связаны следующим соотношением эквивалентности:

$$(a/b)*b + a\%b \equiv a \ (b \neq 0).$$

3.1.4. Аддитивные операторы

При необходимости производятся обычные арифметические преобразования. Уровень приоритета аддитивных операторов равен 4, выполняются эти операторы слева направо.

| Оператор | Название | Типы операндов | Тип результата | Комментарии |
|----------|-----------|--|-----------------------------|---|
| + | Сложение | Арифметические | int, unsigned, long, double | Перед сложением значение целого операнда умножается на величину элемента данных, тип которых соответствует типу указателя |
| | | Один операнд - указатель, другой - интегрального типа (оба операнда не могут быть указателями) | Указатель | |
| - | Вычитание | Арифметические | int, unsigned, long, double | До вычитания значение целого операнда умножается на величину элемента данных, тип которых соответствует типу указателя |
| | | Один операнд - указатель, другой - интегрального типа Операнды - указатели одного типа | Указатель int | |

3.1.4.1. Арифметические действия с указателями. Арифметические действия с указателями отличаются от арифметических действий с обычными целыми значениями и определяются следующим образом. Предположим, что i является целым выражением, указатели p и q указывают на элементы типа T . Сложение значений i и p эквивалентно сложению числа ячеек памяти (в байтах), занятых i элементами типа T . Аналогично определяется операция вычитания. Результат вычитания двух указателей типа $*T$ является не разностью значений двух указателей, а числом элементов типа T , размещенных между ячейками, ссылки на которые обеспечиваются указателями.

Никакие другие арифметические действия с указателями не допускаются. Вычитание указателей имеет смысл только для указателей, ссылающихся на элементы одного и того же массива, поскольку только в этом случае разность адресов элементов массива всегда равна произведению целого значения на величину элемента. В других случаях, даже если указатели одного типа, их разность не будет равной произведению целого на величину элемента.

Более формально арифметические действия с указателями могут быть определены следующим образом:

$$\begin{aligned} p + i &\equiv (T *) ((\text{long}) p + (\text{long}) \text{sizeof}(T) * i) \\ i + p &\equiv (T *) ((\text{long}) p + (\text{long}) \text{sizeof}(T) * i) \\ p - i &\equiv (T *) ((\text{long}) p - (\text{long}) \text{sizeof}(T) * i) \\ p - q &\equiv ((\text{long}) p - (\text{long}) q) / ((\text{long}) \text{sizeof}(T)) \end{aligned}$$

(Для некоторых реализаций языка в приведенных выше определениях слово `long` необходимо заменить словом `unsigned`.)

3.1.5. Операторы сдвига

При необходимости производятся обычные арифметические преобразования. Уровень приоритета операторов сдвига равен 5, порядок выполнения операторов - слева направо.

| Оператор | Название | Тип операнда | Тип результата | Комментарии |
|----------|-------------|--------------|---------------------------------|--|
| $<<$ | Сдвиг влево | Интегральный | Такой же, как у левого операнда | Правый операнд преобразуется к типу <code>int</code> ; левый операнд сдвигается на число разрядов, равное значению |

| | | | | |
|----|-----------------|------------------------|--|--|
| >> | Сдвиг вправо | Инте- граль- ный | Такой же, как у левого операнда | правого операнда; освобождающиеся разряды заполня- ются нулями Правый операнд преобразуется к типу int; левый операнд сдвигается на число разрядов, равное значению правого операнда; сдвиг будет логи- ческим сдвигом, если левый опе- ранд имеет тип unsigned |
|----|-----------------|------------------------|--|--|

3.1.6. Операторы отношения

Если необходимо, производятся обычные арифметические преобразования. Уровень приоритета операторов отношения равен 6, выполняются эти операторы слева направо.

| Оператор | Название | Тип операндов | Тип результата |
|----------|---------------------|---------------------------------|-------------------|
| < | Меньше | Арифметический или указатель | int |
| > | Больше | Арифметический или указатель | int |
| <= | Меньше или равно | Арифметический или указатель | int |
| >= | Больше или равно | Арифметический или указатель | int |

Сравнение указателей является мобильным, если указатели ссылаются на элементы одного и того же массива.

3.1.7. Операторы равенства и неравенства

При необходимости производятся обычные арифметические преобразования. Уровень приоритета операторов равен 7, выполняются они слева направо.

| Оператор | Название | Тип операндов | Тип результата |
|----------|-------------|------------------------------|----------------|
| == | Равенство | Арифметический или указатель | int |
| != | Неравенство | Арифметический или указатель | int |

Единственным целым значением, с которым можно сравнивать указатели, является нулевое значение.

3.1.8 Оператор поразрядное и

Производятся обычные арифметические преобразования. Уровень приоритета оператора равен 8, выполняются такие операторы слева направо.

| Оператор | Название | Тип операндов | Тип результата |
|----------|---------------|---------------|---------------------|
| & | Поразрядное и | Интегральный | int, long, unsigned |

3.1.9. Оператор поразрядное исключающее или

Производятся обычные арифметические преобразования. Уровень приоритета оператора равен 9, порядок выполнения таких операторов - слева направо.

| Оператор | Название | Тип операндов | Тип результата |
|----------|-----------------------------|---------------|---------------------|
| ^ | Поразрядное исключающее или | Интегральный | int, long, unsigned |

3.1.10. Оператор поразрядное *включающее или*

Производятся обычные арифметические преобразования. Уровень приоритета оператора равен 10, порядок выполнения таких операторов - слева направо.

| Оператор | Название | Тип операндов | Тип результата |
|----------|--|---------------|------------------------|
| | Поразрядное <i>включающее или</i> | Интегральный | int, long, unsigned |

3.1.11. Логический (условный) оператор *и*

Уровень приоритета оператора равен 11, выполняются операторы слева направо.

| Оператор | Название | Типы операндов | Тип результата | Комментарии |
|----------|------------------------|------------------------------------|----------------|---|
| && | Логическое <i>и</i> | Арифметический или указатель | int | Если первый операнд равен 0, то результат равен 0; в противном случае результат будет равен 1, если второй операнд не равен 0, и равен 0, если второй операнд равен 0 (если первый операнд равен 0, то второй операнд не вычисляется) |

3.1.12. Логический (условный) оператор *или*

Уровень приоритета оператора равен 12, выполняются такие операторы слева направо.

| Опера- тор | Назва- ние | Тип операн- дов | Тип результата | Комментарии |
|---------------|-------------------|------------------------------------|-------------------|--|
| | Логическое или | Арифметический или указатель | int | Если первый операнд не равен 0, результат равен 1; иначе результат будет равен 1, если второй операнд не равен 0, и равен 0, если второй операнд равен 0 (если первый операнд равен 1, то второй операнд не вычисляется) |

3.1.13. Условный оператор

Выполняются обычные арифметические преобразования. Уровень приоритета оператора равен 13, выполняются такие операторы слева направо.

| Опера- тор | Назва- ние | Типы операн- дов | Тип результата | Комментарии |
|---------------|-------------------|---|---|--|
| ?: | Условный оператор | Арифметические; второй и третий операнды могут быть указателями, структурами, объединениями | int, long, unsigned, double, указатель, структура или объединение | Второй и третий операнды преобразуются к одному и тому же типу |

Условный оператор является единственным оператором, для которого необходимы три операнда; используется он следующим образом:

$$a ? b : c$$

где a , b и c - выражения. Если a не равно 0, то результат выражения $a ? b : c$ равен b ; иначе результат равен c . Из двух последних операндов вычисляется только один. Приведем пример условного выражения:

$$\max = (x > y) ? x : y$$

результат вычисления которого есть максимум из значений x и y . В большинстве языков вышеприведенные вычисления могли бы быть записаны (с использованием синтаксиса Паскаля) как

```
if x > y
then max := x
else max := y
```

что требует от программиста явного присваивания значения переменной (возможно, вспомогательной переменной).

3.1.14. Операторы присваивания

Уровень приоритета операторов равен 14, выполняются такие операторы справа налево.

| Оператор | Название | Типы операндов | Тип результата | Комментарии |
|----------|----------------------|--|---|-------------|
| = | Простое присваивание | Арифметические, указатели, объединения или структуры | Если оба операнда имеют арифметический тип, то значение правого операнда преобразуется к типу левого операнда | |

| | | | | |
|---------|------------------------------|-----------------------------|-----------------------------|---|
| <знак>= | Сложное присваи- вание | См. объ- яснение ниже | См. объ- яснение ниже | <знак> обозначает один из знаков +, -, *, /, %, >>, <<, &, ^ или (смысл обозначений объ- ясняется ниже) |
|---------|------------------------------|-----------------------------|-----------------------------|---|

В результате выполнения оператора

$$v = e$$

где v - переменная, а e - выражение, значение выражения становится новым значением переменной v .

Оператор присваивания

$$v \text{ <знак> } = e$$

приблизительно эквивалентен оператору присваивания

$$v = v \text{ <знак> } e$$

Типы операндов и результата сложного оператора присваивания можно определить на основании этой эквивалентности.

Однако приведенный выше эквивалент для сложного оператора присваивания не совсем точен - в выражении

$$v \text{ <знак> } = e$$

операнд v вычисляется только один раз, в то время как в выражении

$$v = v \text{ <знак> } e$$

этот операнд вычисляется дважды. Это различие проявляется в побочных эффектах, связанных с вычислением операнда v , например, при изменении значения какой-либо переменной. Рассмотрим оператор присваивания

$$a[i++] *= n$$

при выполнении которого вычисление левого операнда дает побочный эффект - увеличение значения переменной i . Следовательно, это присваивание не эквивалентно присваиванию

```
a[i++] = a[i++] * n
```

Эквивалентом первого оператора присваивания может служить последовательность операторов

```
a[i] = a[i] * n  
i = i + 1
```

а эквивалентом второго - последовательность операторов

```
a[i] = a[i+1] * n  
i = i + 2
```

или последовательность операторов

```
a[i+1] = a[i] * n  
i = i + 2
```

в зависимости от того, какая часть оператора присваивания вычисляется раньше - левая или правая; порядок таких вычислений не определен.

3.1.15. Оператор запятая

Уровень приоритета оператора равен 15, выполняются такие операторы слева направо.

| Оператор | Название | Тип результата | Комментарии |
|----------|----------|------------------------------------|---|
| , | Запятая | Совпадает с типом правого операнда | Объединяет два выражения в одно выражение, значением которого является значение правого операнда; значение левого операнда вычисляется только для получения побочных эффектов |

В контексте, где запятая используется для других целей, например для отделения аргументов функции, выражения, включающие оператор запятая, должны заключаться в скобки.

3.1.16. Таблица приоритетов и порядка выполнения операторов

Операторы таблицы перечислены в порядке убывания приоритета.

| Приоритет | Операторы | Обозначение | Порядок |
|-----------|------------------------------------|---------------------------|---------------|
| 1 | Вызова функций или выбора | () [] . -> | Слева направо |
| 2 | Унарные | * & - ! ~ ++ -- sizeof | Справа налево |
| 3 | Мультипликативные | * / % | Слева направо |
| 4 | Аддитивные | + - | Слева направо |
| 5 | Сдвига | << >> | Слева направо |
| 6 | Отношения | < > <= >= | Слева направо |
| 7 | Равенства и неравенства | == != | Слева направо |
| 8 | Поразрядное и | & | Слева направо |
| 9 | Поразрядное <i>исключающее</i> или | ^ | Слева направо |
| 10 | Поразрядное <i>включающее</i> или | | Слева направо |
| 11 | Логическое и | && | Слева направо |
| 12 | Логическое <i>или</i> | | Слева направо |
| 13 | Условный | ?: | Справа налево |
| 14 | Присваивания | = <знак>= | Справа налево |
| 15 | Запятая | , | Слева направо |

3.2. ВЫРАЖЕНИЯ

Выражениями называются компоненты программы, составленные с использованием операторов, литералов, констант, переменных (включая массивы, структуры и объединения) и вызовов функций. Порядок вычисления выражения определен лишь требованиями соответствия семантике операторов и соблюдения правил приоритета и порядка выполнения операторов. При выполнении этих требований компилятор свободен в выборе порядка вычисления выражения, даже если вычисление подвыражений может привести к побочным эффектам.

В отличие от большинства других языков, в языке Си для задания определенного порядка вычисления выражения недостаточно только соответствующей расстановки скобок, так как компилятор может произвольно переупорядочивать выражения, включающие ассоциативные и коммутативные операторы (*, +, |, ^) даже при наличии скобок. Для задания желаемого порядка выполнения выражения нужно использовать дополнительные присваивания, если требуется, с использованием временных переменных.

Необходимо с осторожностью использовать выражения, при вычислении которых возможны побочные результаты, так как результаты вычисления таких выражений часто проявляются не сразу и, кроме того, зависят от используемого компилятора. Например, в результате вычисления операторов присваивания

```
j = 3;  
i = (k = j + 1) + (j = 5);
```

значение переменной *i* будет равно 9 или 11 в зависимости от того, какое подвыражение второго оператора будет вычислено первым. Таким образом, с использованием разных компиляторов можно получить различные результаты.

3.2.1. Постоянные выражения

Постоянными выражениями называются выражения, сформированные с использованием констант типов `integer`, `char` и `enum`, оператора `sizeof`, унарных операторов - и ~, бинарных операторов

+ - * / % & | ^ << >> == != < > <= и >=

и тернарного оператора ?..

Постоянные выражения используются в операторе `switch` (с последующими операторами `case`), в инициализаторах границ массивов и в операторе препроцессора `#if` (см. гл. 9). С некоторыми ограничениями в инициализаторах может быть также

использован оператор адресации &. Оператор sizeof и перечисляемые константы в операторе препроцессора #if не допускаются.

3.3. ЗАДАЧИ

1. Логические операторы *или* и *и* (*||* и *&&*) являются условными логическими операторами (второй операнд вычисляется только при необходимости). В других языках программирования, таких как Паскаль и Фортран, в логических операторах всегда вычисляется значение обоих операндов, даже если результат может быть определен вычислением одного операнда. Каковы "за" и "против" условных и безусловных логических операторов?

2. В отличие от Фортрана, в самом языке Си отсутствует оператор возведения в степень, однако функция возведения в степень имеется в библиотеке math (см. приложение А). Приведите соображения, объясняющие, почему оператор возведения в степень не включен в язык. В чем, с точки зрения пользователя, состоит недостаток такого решения (предположим, что пользователь не имеет доступа к библиотеке math)? Предположим, что нужно вычислить x^y (для некоторого неотрицательного целого y). Очевидный, но неэффективный способ реализации возведения в целочисленную степень - многократное умножение. Существует более эффективный алгоритм, приведенный ниже:

```
a = x; b = y; z = 1; /* Вначале  $z a^b = x^y$  */
                      /* в конце алгоритма */
                      /*  $z = x^y$  */
```

```
while ( b != 0 ) {
    if ( odd(b) ) {
        z = z * a;
        b--;
    }
    else {
        a = a * a;
        b = b / 2;
    }
}
/* результат - значение "z" */
```

Операция odd(x) возвращает 1, если x - нечетное число, и 0, если четное. Подумайте о том, как реализовать операцию odd.

Вы уверены, что этот алгоритм будет работать? Для доказательства его работоспособности можно воспользоваться тем, что программа пытается сохранить равенство $z a^b = x^y$ при значении b, стремящемся к 0. По окончании алгоритма значение b

будет равно 0 и тогда приведенное выше соотношение превратится в равенство $z = x^y$.

Глава 4

Операторы управления

Операторы управления языка Си принципиально ничем не отличаются от операторов управления таких языков, как Паскаль или Ада. Эти операторы соответствуют принципам структурного программирования [13] и представляют собой, в основном, конструкции с одним входом и одним выходом. Хотя в языке Си имеется оператор *goto* (считается, что его использование затрудняет чтение и понимание текста программы [14]), в язык введены также операторы *break* и *continue*, обеспечивающие управляемые переходы. Во всех случаях, когда это возможно, вместо оператора *goto* должен использоваться оператор *break* или *continue*. В отличие от языка Си, в таких языках, как Фортран или Паскаль, отсутствует механизм управляемых переходов.

4.1. ВЫРАЖЕНИЯ И ОПЕРАТОРЫ

Любое выражение может быть преобразовано в оператор добавлением к нему точки с запятой. Запись вида

выражение;

является оператором. Значение *выражения* игнорируется. Действие такого оператора состоит в создании побочного эффекта вычислением значения *выражения*.

Следующие два оператора, оператор присваивания и оператор вызова функции (используемый для функции, не возвращающей значения), получены добавлением точки с запятой к соответствующим выражениям, они являются специальными и будут рассмотрены отдельно.

4.2. ПУСТОЙ ОПЕРАТОР

Пустой оператор обозначается точкой с запятой:

;

Его использование необходимо в тех случаях, когда логически не требуется выполнения каких-либо действий, но в соответствии с правилами синтаксиса присутствие оператора обяза-

тельно. Например, пустой оператор используется в качестве тела следующего цикла *while*:

```
while ((c = getchar()) == BLANK)
    ;
```

в котором делается переход к первому знаку, отличному от пробела (здесь BLANK - константа, определенная пользователем).

4.3. СОСТАВНОЙ ОПЕРАТОР

Составной оператор используется в следующих случаях:

1. Чтобы сгруппировать несколько логически связанных операторов в один оператор.
2. В качестве тела функции.
3. Для ограничения видимости определений частью программы, т.е. для локализации действия описаний.

Составной оператор имеет следующую форму:

```
{
    определения и описания
    операторы
}
```

Определения переменных внутри составного оператора имеют больший приоритет, чем определения переменных с тем же именем для области действия составного оператора. Эти переменные видимы (доступны) только внутри составного оператора. Глобальные переменные являются видимыми внутри составного оператора только при условии, что их определения не изменены локальными определениями.

4.4. ОПЕРАТОР ПРИСВАИВАНИЯ

Оператор присваивания имеет следующую форму:

переменная = *выражение*;

где *переменная* является выражением - ссылкой на объект в памяти. Например:

```
i = i + 1;  
*pc = 'c';
```

Оператор присваивания является выражением вида

переменная = выражение

преобразованным в оператор добавлением к нему точки с запятой. Значением этого выражения является значение, присвоенное переменной. Следовательно, используя последовательность операторов присваивания в одном операторе, можно присвоить значения сразу нескольким переменным, например:

```
i = j = k = 0;
```

Оператор присваивания может быть также образован из составных операторов присваивания. Такой оператор присваивания имеет следующую форму:

переменная <знак>= выражение;

где <знак> обозначает один из знаков

+ - * / % >> << & ^ |

(которые уже рассматривались в разд. 3.1.14).

4.5. ОПЕРАТОР *if*

Оператор *if* имеет две формы:

if (выражение) оператор₁;

и

```
if (выражение)  
    оператор1;  
else  
    оператор2;
```

Если в результате вычисления значения *выражения* получено значение *истина* (ненулевое значение), то в обеих формах оператора *if* выполняется *оператор₁*. Если вычисленное значение выражения равно значению *ложь* (нулевое), тогда выполне-

ние оператора *if*, представленного в первой форме, заканчивается, а в операторе, имеющем вторую форму, выполняется оператор₂.¹

Совместное использование обеих форм оператора *if* приводит к неоднозначности, называемой "проблемой висящего *else*". Например, вложенный оператор *if*

```
if (e1) if (e2) sa; else sb;
```

может быть интерпретирован² как

```
if (e1)  
  if (e2)  
    sa;  
else  
  sb;
```

или как

```
if (e1)  
  if (e2) sa;  
else  
  sb;
```

Эта неоднозначность разрешается в языке Си с помощью правила, в соответствии с которым часть *else* оператора всегда относится к синтаксически самому правому (игнорируя любые отступы) оператору *if* без части *else*. Следовательно, первая интерпретация является интерпретацией, принятой в языке Си.

Существует простой способ, позволяющий избавиться от такой неоднозначности: следует избегать одновременного использования обеих форм операторов *if* в конструкциях с вложенными операторами *if*. При необходимости можно воспользоваться пустым оператором. Например, вторая интерпретация вышеприведенного оператора *if* может быть записана как

¹ Синтаксически как оператор₁, так и оператор₂ должен быть единственным оператором. Следовательно, если в этом месте должно выполняться более одного оператора, то эти операторы необходимо объединить в один оператор, заключив их в фигурные скобки, т.е. использовать конструкцию *составной* оператор.

² Эти две интерпретации показаны с помощью соответствующих отступов. На восприятие текста компилятором языка Си, в отличие от человеческого восприятия, отступы не влияют.

```

if (e1)
  if (e2)
    sa;
  else
    ;      /* точка с запятой здесь обозначает */
          /* пустой оператор */
else
  sb;

```

Для явного указания намерений программиста можно использовать и фигурные скобки. Например, обе вышеприведенные интерпретации можно записать явно как

```

if (e1) {
  if (e2)
    sa;
  else
    sb;
}

```

и

```

if (e1) {
  if (e2) sa;
}
else
  sb;

```

4.6. ОПЕРАТОР *switch*

Оператор *switch* (переключатель) используется для разветвления программы по нескольким направлениям. Он имеет следующую форму:

```

switch (e) {
case ce1: s1; break;
case ce2: s2; break;
.
.
.
case cek: sk; break;
default: sk+1;
}

```

где

1. e - целое выражение (или выражение, которое может быть преобразовано в целое выражение);
2. ce_i - целое выражение с постоянным значением (или выражение, которое может быть преобразовано к такому выражению);
3. s_i обозначает операторы, число которых может быть больше или равно нулю.

Метки ce_i , обозначающие альтернативы *case*, должны быть уникальными; двух одинаковых меток быть не может. Только одна альтернатива может получить префикс *default*.

Результатом выполнения оператора *switch* является выбор альтернативы с меткой ce_i , которая равна значению выражения переключателя e ; в этом случае выполняются операторы s_i . В случае, если выражение переключателя не равно ни одному из выражений альтернатив *case*, то выполняются операторы s_{k+1} ; при отсутствии альтернативы *default* не выполняется ни одна из альтернатив оператора *switch*.

Ниже приведен пример использования оператора *switch* в программном сегменте простого интерпретатора польской записи:

```
switch (c) {
case '+': сложить; break;
case '-': вычесть; break;
case '*': умножить; break;
case '/': разделить; break;
default: занести в стек;
}
```

Если действие двух или более альтернатив совпадает, то эти альтернативы могут быть объединены с помощью нескольких префиксов для альтернативы *case* с оператором s_i :

```
switch (e) {
case  $ce_{11}$ : case  $ce_{12}$ : ... : case  $ce_{1n1}$ :  $s_1$ ; break;
case  $ce_{21}$ : case  $ce_{22}$ : ... : case  $ce_{2n2}$ :  $s_2$ ; break;
.
.
.
case  $ce_{k1}$ : case  $ce_{k2}$ : ... : case  $ce_{knk}$ :  $s_k$ ; break;
default:  $s_{k+1}$ ;
}
```

В проиллюстрированных двух формах оператора *switch* альтернативные действия s_i из-за присутствия оператора *break* после каждого из них являются взаимоисключающими. Употребление оператора *break* в данном случае не обязательно; если он отсутствует, выполнение программы будет продолжаться от од-

ной альтернативы к другой. Однако, чтобы программа была более ясной и надежной, а также для упрощения ее модификации настоятельно рекомендуется использование оператора *break* [20]. Альтернатива *default* в языке Си также не обязательна, тем не менее для улучшения ясности программы рекомендуется ее использовать даже в том случае, когда предписываемое ею действие заключается всего лишь в выполнении пустого оператора, который мог быть просто опущен.

4.7. ЦИКЛЫ

Существует три вида циклов: *while*, *for* и *do*.

4.7.1. Цикл *while*

Цикл *while* имеет следующую форму:

```
while (e) s;
```

Оператор *s* выполняется до тех пор, пока значение выражения *e* равно "истина"; значение *e* вычисляется перед каждым выполнением оператора *s*.

4.7.2. Цикл *for*

Оператор цикла *for*

```
for (e1; e2; e3) s;
```

является удобной сокращенной записью для цикла *while* вида

```
e1;  
while (e2) {  
    s;  
    e3;  
}
```

Выражение *e*₁ служит для задания начальных условий выполнения цикла, выражение *e*₂ обеспечивает проверку условия выхода из цикла, а выражение *e*₃ модифицирует условия, заданные выражением *e*₁. Любое из выражений *e*₁, *e*₂ или *e*₃ может быть опущено. Если опущено *e*₂, то по умолчанию вместо него подставляется значение TRUE.

Например, цикл *for*

```
for (; e2; ) s;
```

с опущенными выражениями e_1 и e_2 эквивалентен циклу

```
while ( $e_2$ )  $s$ ;
```

Цикл *for*

```
for (;;)  $s$ ;
```

со всеми опущенными выражениями эквивалентен циклу

```
while (TRUE)  $s$ ;
```

т.е. эквивалентен бесконечному циклу. Такой цикл может быть прерван только явным выходом из него с помощью операторов *break*, *goto* или *return*, содержащихся в теле цикла s .

Несмотря на внешнее сходство с итеративными циклами *for* языков Паскаль и Ада или итеративными циклами *do* языков Фортран или ПЛ/1, цикл *for* языка Си не является их семантической копией. Цикл *for* языка Си обладает большей общностью, чем циклы *for* и *do* других языков; однако в отличие от этих циклов, в общем случае число итераций в цикле *for* языка Си не может быть определено до выполнения этого цикла.

Цикл *for* и его альтернатива цикл *while* семантически почти эквивалентны, но, как указывается в работе [77], не идентичны. Например, рассмотрим случай, когда оператор s является оператором *continue* или составным оператором, содержащим оператор *continue*. Действие оператора *continue* состоит в переходе к концу цикла, что имеет различные последствия для цикла *for* и его эквивалента в форме цикла *while*. В случае с циклом *for* выражение e_3 выполняется до вычисления значения выражения e_2 , в то время как в эквивалентном цикле *while* выражение e_3 пропускается.

4.7.3. Цикл *do*

Цикл

```
do оператор while ( $e$ );
```

выполняется до тех пор, пока выражение e имеет значение "истина". В отличие от цикла *while*, в котором проверка условия окончания цикла делается до выполнения тела цикла, в цикле *do* такая проверка имеет место после выполнения тела цикла. Следовательно, тело цикла *do* будет выполнено хотя бы один раз, даже если выражение e имеет значение "ложь" с самого начала. Цикл *do* аналогичен циклу *repeat* в языке Паскаль, отличаясь от него лишь тем, что цикл *repeat* выполняется до тех

пор, пока некоторое условие выхода из цикла не становится истинным, а цикл *do* выполняется все время, пока некоторое условие остается истинным.

4.8. ОПЕРАТОР *break*

Оператор *break* используется для выхода из оператора *while*, *do*, *for* или *switch*, непосредственно его содержащего. Управление передается на оператор, следующий за оператором, из которого осуществлен выход. Оператор *break* имеет форму

```
break;
```

4.9. ОПЕРАТОР *continue*

Оператор *continue* служит для пропуска оставшейся части выполняемой итерации цикла, непосредственно его содержащего. Если условиями цикла допускается новая итерация, то она выполняется, в противном случае цикл завершается. Оператор *continue* имеет следующую форму:

```
continue;
```

Действие оператора *continue* в приведенных ниже примерах эквивалентно действию оператора *goto* (см. разд. 4.12), предписывающего переход на метку *next*.¹

```
while ( ... ) {  
    ...  
    next;;  
}
```

```
do {  
    ...  
    next;;  
} while ( ... );
```

```
for ( ... ) {  
    ...  
    next;;  
}
```

¹ Предполагается, что оператор *continue* не находится внутри вложенного цикла.

4.10. ОПЕРАТОР ВЫЗОВА ФУНКЦИИ

Оператор вызова функции имеет следующую форму:

имя-функции(список фактических параметров);

Этот оператор используется для вызова функций, не возвращающих значений в вызывающую программу¹, или функций, результат выполнения которых должен быть игнорирован. Более детально функции обсуждаются в гл. 5.

4.11. ОПЕРАТОР *return*

Оператор *return* используется в функции для возвращения ее результата в вызывающую программу и окончания выполнения функции. Оператор *return* имеет две формы:

return e;

где *e* - выражение, представляющее результат работы функции, и форму

return;

Функция может иметь несколько операторов *return* или не иметь их совсем. Первая форма оператора *return* должна использоваться только в функциях, возвращающих значение, вторая форма предназначена только для функций, не возвращающих значения. Более детально оператор *return* обсуждается в гл. 5.

4.12. ОПЕРАТОР *goto*

Оператор *goto* предназначен для безусловной передачи управления к оператору с указанной меткой. Он имеет следующую форму:

goto метка;

¹ Функции, не возвращающие значения, называются в языке Фортран подпрограммами, а в языках Паскаль и Ада - процедурами.
Выражение

имя-функции(список фактических параметров)

используется для вызова функций, не возвращающих значений; это выражение может быть частью другого выражения.

Неограниченное использование оператора *goto* приводит к ухудшению понимания программы и поэтому не рекомендуется [14]. Следовательно, каждое употребление оператора *goto* должно быть обосновано программистом.

4.13. МЕТКИ ОПЕРАТОРОВ

Перед каждым оператором программы, написанной на языке Си, может быть поставлена метка:

метка: оператор

где *метка* является идентификатором. Операторы помечаются таким образом, чтобы на них можно было делать ссылки в операторах *goto*.

4.14. ЗАДАЧИ

1. Определите результат выполнения следующей программы:

```
int i, test[2];

i = 0;
test[i] = i = i + 1;
/* несколько одновременных присваиваний */
/* с побочным эффектом */
```

Сопоставьте ваш ответ с результатом выполнения тестовой программы, полученной с помощью вашего компилятора.

2. Напишите вложенные операторы *if*, эквивалентные оператору *switch*, в общей форме, представленной выше. Является ли оператор *if* более мощным, чем оператор *switch*?

Использование операторов *break* в каждой альтернативе оператора *switch* не требуется. Если бы некоторые (или все) операторы *break* были опущены, легко было бы смоделировать этот вариант оператора *switch* с помощью операторов *if*?

3. Предложите вариант синтаксиса для оператора *if*, который решал бы проблему неоднозначности для вложенных операторов *if* обеих форм.

4. Почему в операторе *switch* для меток оператора *case* необходимы выражения с целыми постоянными значениями? Что случилось бы, если вместо целочисленных выражений с постоянными значениями были бы разрешены общие целые выражения? Оцените как удобства пользователя, так и аспекты реализации.

5. Циклы *for* в языках Фортран, Паскаль и Ада всегда завершаются; цикл *for* в языке Си более общий и может не завер-

шаться. Какие ограничения нужно наложить на форму цикла *for* в языке Си, чтобы он всегда завершался?

Глава 5

Функции и завершенные программы

5.1. ФУНКЦИИ

Абстракция управления в языке Си обеспечивается с помощью функций. Все функции могут быть рекурсивными. В языке Си отсутствуют подпрограммы (процедуры), однако возврат функцией значения в вызывающую программу не обязателен. Следовательно, функции могут быть разделены на две категории - функции, возвращающие значения, и функции, не возвращающие значения в вызывающую программу (подпрограммы).¹

Определение функций, возвращающих значение, имеет следующий формат:²

```
[static] тип-результата или я-функции (формальные параметры)  
описания формальных параметров  
{  
    тело функции  
}
```

где *или я-функции* - описатель (см. гл. 2), а тело функции имеет вид

*определения и описания
операторы*

В качестве результата функция не может возвращать массив или функцию, но может возвращать указатель на массив или функцию. Выполнение функции, возвращающей значения, обязательно завершаться оператором `return` вида

¹ Такое деление функций на две категории в [77] не предлагалось, однако использование различных форм для этих двух категорий функций может сделать программу более ясной и легко читаемой.

² Указание типа-результата функции в языке Си не является обязательным. Если тип результата не указан, то предполагается, что результат имеет тип `int`. Поскольку указание типа функции приводит к большей ясности и легкости чтения программы, а также упрощает нахождение в ней ошибок, тип функции всегда должен быть указан явно.

```
return e;
```

который обеспечивает выдачу результата *e*. Функция, возвращающая значение, может содержать более одного оператора `return`.

Определения функции, не возвращающей значения, имеют следующий формат:¹

```
[ static ] void имя-функции(формальные параметры)  
описания формальных параметров  
{  
    тело функции  
}
```

Выполнение такой функции завершается, если выполнено ее тело или оператор `return` вида

```
return;
```

Функция, не возвращающая значения, может содержать более одного оператора `return`.

Класс памяти `static` (необязательный) ограничивает видимость функции и других внешних определений. Функция с классом памяти `static` невидима вне содержащего ее файла (см. разд. 5.1.2).

Ниже приведен пример простой функции, находящей максимум из двух своих параметров:

```
int max(a, b)  
int a, b;  
{  
    return (a>b) ? a:b;  
}
```

Если в тексте программы есть обращение к функции, то необходимо описание функции, которое в тексте должно быть помеще-

¹ Как уже упоминалось, тип `void` добавлен в язык Си недавно. Для компиляторов, не способных обрабатывать этот тип, программист может определить тип `void` как

```
#define void    int
```

и использовать его для определения функций, не возвращающих значения. Рекомендуется следовать этому соглашению для улучшения ясности программы. Однако в таких случаях компилятор будет не в состоянии обнаружить некорректное использование этих функций для возврата значений, поскольку на самом деле рассматриваемые функции возвращают значения, и эти значения имеют тип `int`.

но раньше ее определения. Описания функции имеют следующую форму:

```
[static | extern] тип-результата или-функции();  
[static | extern] void или-функции();
```

Если в описании не указан класс памяти, то, по умолчанию, предполагается `extern`. Ниже приведены несколько примеров описаний функций:

```
int max();  
extern char *strcpy();  
char *malloc();
```

5.1.1. Описания параметров

Описания параметров аналогичны описаниям обычных переменных, за исключением того, что единственным классом памяти, который может быть указан в таких описаниях, является класс `register`. В описании формальных параметров - массивов может быть опущено максимальное значение первого индекса массива, например:

```
int a[], b[4], d[][4], e[2][4];  
/* допустимые описания параметров - массивов */
```

Описание параметра

```
int c[][]; /* неверное описание параметра - массива */
```

является неправильным, так как в нем не указано максимальное значение второго индекса.

5.1.2. Управление видимостью функций

В языке Си вложенные функции не допускаются. Следовательно, невозможно определить функции, локальные по отношению к функции, а именно, невозможно определить функции, которые были бы невидимы вне функции, их содержащей. Это ограничение на вложенность функций для большинства практических ситуаций не является недостатком, поскольку видимостью функций можно управлять с помощью атрибута `static`. Например, пусть необходимо гарантировать, чтобы к функции `a` доступ был возможен только из функции `b`. Тогда эти две функции могут быть определены в отдельном файле с атрибутом `static`:

```
static int a(...)
{
.
.
.
}

int b(...)
{
.
.
.
}
```

Функция *a* может быть вызвана из функции *b*, но никакая другая функция (физически размещенная в другом файле) не сможет вызвать функцию *a*.

5.1.3. Вызов функций

Вызов функции, возвращающей значения, имеет следующую форму:

имя-функции(список фактических параметров)

где фактические параметры¹ могут быть выражениями. Например:

```
a = max(a, 5);
max(max(a, 5), e)
getchar()
fs = fopen(argv[1], "r");
```

Функции, не возвращающие значения, вызываются аналогично, но их вызовы преобразуются в следующие операторы:

имя-функции(список фактических параметров);

¹ Параметры, указанные в заголовке функции, называются формальными параметрами. Параметры, заданные в вызове функции, называются фактическими параметрами. Как формальные, так и фактические параметры будут называться просто параметрами; прилагательные "формальный" и "фактический" будут использоваться только в тех случаях, когда из контекста неясно, о каких параметрах идет речь.

В некоторых языках программирования, например в языке Фортран, используется другая терминология; фактические параметры называют аргументами, а формальные - параметрами.

Например:

```
delay(0.5);  
least_square(x, y, n, &a, &b);  
partition(a, 1, u, &i, &j);  
quicksort(a, 1, j);
```

Передача параметров обычно реализуется двумя способами: *передачей по значению* и *передачей по ссылке*. В языке Си параметры передаются только одним способом - *по значению*. При передаче параметров по значению до выполнения функции значения фактических параметров копируются в формальные параметры. При передаче параметров по ссылке формальные параметры фактически становятся синонимами для фактических параметров. Если фактический параметр является выражением, то соответствующий формальный параметр становится синонимом для временной переменной, содержащей значение этого выражения.

Передача параметров по значению удобна, когда функция возвращает значение без изменения своих фактических параметров. Однако если функция должна изменять значения своих фактических параметров, то должен использоваться другой механизм передачи параметров, такой как передача параметров по ссылке. Передача параметров по значению может также оказаться неэффективной, если она требует большого объема копирования данных; в этом случае более приемлемой была бы передача параметров по ссылке. В языке Си передача параметров по ссылке моделируется передачей в качестве параметра указателя на соответствующий объект. Сама по себе передача указателей на объекты не является серьезным неудобством, но в вызываемой функции эти объекты должны обрабатываться иначе, чем объекты, передаваемые по значению.

Существует одно исключение из правила, в соответствии с которым все параметры передаются по значению - имя массива можно рассматривать как указатель на массив, откуда следует, что передача массива выполняется по ссылке.

5.1.4. Обращение к функции до ее определения

Функция может быть вызвана или передана как параметр до того, как ее определение встретится в тексте программы при условии, что до обращения к ней в тексте встретилось ее описание.¹ Описание функции указывает только тип результата

¹ Определение функции может быть и в другом файле (см. гл. 6).

функции и, что необязательно, ее класс памяти. Это требование необходимо для генерации кода и ограниченной проверки ошибок.

Ниже приведен пример, иллюстрирующий использование функции до того, как в тексте программы встретится ее определение.

```
void add();
int in_table();
.
.
.
main()
{
    ...
    if (!in_table(a))
        add(a, it);
    ...
}
.
.
.
void add(x, it) /* добавить элемент "a", который */
               /* имеет тип "it", к таблице      */
               /* символов                        */
char *x;
item_type it;
{
    ...
}

int in_table(x)
char *x;
{
    ...
}
```

Вызовы функций `add` и `in_table` предшествуют их определениям в тексте программы, но описания этих функций

```
void add();
int in_table();
```

даны раньше их вызовов.

5.1.5. Пример, иллюстрирующий передачу параметров

Проиллюстрируем теперь различие между передачей параметров по значению и по ссылке. Рассмотрим функцию `swar`; предполагается, что она изменяет значения своих параметров:

```
/* начальная версия функции "swar" */
void swar(a, b)
int a, b;
{
    int temp;

    temp = a;
    a = b;
    b = temp;
}
```

Вызов этой функции не оказывает влияния на ее фактические параметры, т.е. вызов

```
swar(x, y);
```

не оказывает влияния на значения `x` и `y` (которые являются переменными типа `int`). Значения переменных `x` и `y` копируются в формальные параметры `a` и `b` соответственно, и уже эти значения формальных параметров затем изменяются. Исходные значения переменных `x` и `y` остаются неизменными.

Если функция `swar` должна изменять значения своих фактических параметров, ее необходимо модифицировать. Естественно предположить, что эта версия функции `swar` должна вместо значений получать адреса переменных, значения которых должны изменяться:

```
/* исправленная версия функции swar */
void swar(a, b)
int *a, *b;
{
    int temp;

    temp = *a;
    *a = *b;
    *b = *temp;
}
```

Необходимо также изменить обращение к функции `swar` так, чтобы ей передавались не значения изменяемых переменных, а их адреса. В приведенном ниже вызове функции иллюстриру-

ется применение оператора адресации &, используемого для определения адресов переменных x и y:

```
swap(&x, &y);
```

Отметим различие в двух версиях функции swap: когда в качестве фактических параметров передаются адреса, необходимо использовать оператор косвенной ссылки *. В языках, допускающих передачу параметров по ссылке, при такой передаче параметров нет необходимости в операторе косвенной ссылки.

5.1.6. Автоматические преобразования фактических параметров

В процессе передачи параметров возможны следующие преобразования типов:

| Тип фактического параметра | Преобразование |
|----------------------------|--|
| float | Все фактические параметры типа float перед их передачей функции преобразуются к типу double. Следовательно, тип всех формальных параметров меняется на double |
| char и short int | Все фактические параметры типов char и short int преобразуются к типу int. Следовательно, тип всех формальных параметров меняется на int |
| массивы | Имя массива является фактически указателем на первый элемент этого массива. Следовательно, при использовании имени массива в качестве фактического параметра это имя является адресом массива, который и передается. Описания всех формальных параметров-массивов изменяются так, чтобы они стали указателями ¹ |

¹ Одно из следствий такого изменения заключается в том, что формальные параметры - массивы становятся переменными, и их значения могут быть изменены программистом. С другой стороны, локальные или внешние имена массивов являются константами и их значения не могут быть изменены программистом.

5.1.7. Передача функций в качестве параметров

Функции могут передаваться в качестве параметров. Как и в случае массивов, фактически передаваемое значение является адресом функции. В качестве примера рассмотрим функцию `cmp`, сравнивающую два элемента типа структуры `employee` и возвращающую значения `TRUE` или `FALSE` в зависимости от значений элементов:

```
int cmp();
```

Функция `cmp` в качестве параметра передается функции `sort`, которая сортирует массивы с элементами типа `employee`; обращение к функции `sort` имеет следующий вид:

```
sort(emp, n, cmp);
```

где `emp` - сортируемый массив, `n` - размерность массива и `cmp` - функция для упорядочения элементов, как уже упоминалось выше.

Определение функции `sort` может иметь следующий вид:

```
void sort(a, n, fp)
employee a[];
int n;
int (*fp)(); /* "fp" указывает на */
              /* функцию сравнения */
{
    .
    .
    .
    if ((*fp)(a[i], a[j]) > 0)
        /* иллюстрируется использование "fp" */
    .
    .
    .
}
```

Возможность передачи функций в качестве параметров может оказаться очень полезной. Например, предположим, что вызов

```
sort(emp, n, greater);
```

обеспечивает сортировку массива `emp` в возрастающем порядке, где функция `greater(x, y)` возвращает значение `TRUE`, если в соответствии с некоторым правилом упорядочения `x` больше `y`, и возвращает значение `FALSE`, если `x` меньше `y`. Рассмотрим

теперь функцию `smaller(x, y)`, возвращающую значение `TRUE` при `x` меньше `y` (в соответствии с тем же правилом упорядочения, что и для функции `greater`), и возвращающую значение `FALSE` в противоположном случае. Тогда вызов

```
sort(emp, n, smaller);
```

обеспечит сортировку массива `emp` в убывающем порядке.

Используя в качестве параметра функции `sort` функцию сравнения, мы можем сортировать массив - фактический параметр в соответствии с любым желаемым правилом упорядочения. Это позволяет нам написать функцию сортировки общего назначения, которая эквивалентна целому семейству функций сортировки.

5.1.8. Спецификации функции

Информация о функции, обеспечивающая интерфейс с пользователем, называется *спецификацией функции* и состоит, по соглашению, из трех частей:

1. Списка операторов *include*, указывающего файлы, которые должны быть включены в текст функции. Обычно эти файлы содержат описания, необходимые для использования функции.

2. Описания функции.

3. Описания параметров функции.

Каждая спецификация функции содержит по крайней мере описание функции. Если функция имеет параметры, то спецификация функции также содержит описания параметров.

Ниже приведен пример спецификации функции:

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

```
int putc(c, stream)
```

```
char c;
```

```
FILE *stream;
```

Цель спецификации функции - обеспечение пользователя достаточной информацией, чтобы

- 1) включить в функцию все файлы, необходимые для ее использования;

- 2) описать переменные, которые будут использоваться как фактические параметры или как компоненты фактических параметров;

- 3) описать переменные, которые необходимы для хранения результатов работы функции;

- 4) использовать функцию в выражениях в соответствии с ее типом;

5) писать синтаксически правильные вызовы функции так, чтобы компилятор языка Си или программа проверки типов `lint` не сообщала о неправильном использовании функции.

Спецификация функции не дает какой-либо содержательной информации о назначении функции, кроме той, которая следует из имени функции и ее типа, а также из типов и имен ее параметров.

5.2. ЛЕКСИЧЕСКАЯ ОБЛАСТЬ ДЕЙСТВИЯ ИДЕНТИФИКАТОРОВ

Лексической областью действия идентификатора называется область программы, в которой его описание или определение имеет силу. Лексическая область действия

внешнего идентификатора располагается от его описания или определения до конца файла, его содержащего; формального параметра или функции - сама эта функция; идентификатора, определенного в блоке, - этот блок; метки - функция, в которой она описана.

Определение идентификатора в блоке имеет больший приоритет, чем любое определение того же самого идентификатора, данное вне этого блока, т.е. локальное определение имеет больший приоритет, чем глобальное.

В качестве примера, иллюстрирующего лексическую область действия идентификаторов, рассмотрим файл с именем `score.c`, содержащий следующую программу:

```
extern float lower, upper;
static float accuracy;
float f();
double fabs();

float integrate(fp, a, b, eps)
float (*fp)(), a, b, eps;
{
    float sum;

    ... fabs(...) ...
}

main()
{
    ... integrate(f, lower, upper, accuracy) ...
}

float f(a)
```

```
float a;
{
    float accuracy;

    ...
}
```

Область действия идентификаторов `lower`, `upper`, `accuracy` (определенных вне функции `f`), `f`, `fabs`, `integrate` и `main` располагается от их описаний или определений (что из них встретится раньше) до конца файла `score.c`. Однако поскольку функция `f` содержит локальный идентификатор `accuracy`, то имеет силу второе определение переменной `accuracy` в теле функции `f`, а не первое.

Поскольку областью действия параметра функции является соответствующее тело функции, существование в функциях `integrate` и `f` параметра с одним и тем же именем `a` не вызывает конфликта.

Заметим, что оказалось необходимым дать описания функций `f` и `fabs`, так как в файле `score.c` имеется обращение к ним до того, как встречены их определения. Определение функции `fabs` отсутствует в файле `score.c` и для успешного выполнения всей программы должно содержаться в каком-либо другом файле. Если бы определение функции `f` встретилось до всех обращений к этой функции, то отпала бы необходимость в ее описании, т.е. создалась бы точно такая же ситуация, как и в случае функции `integrate`.

5.3. ВВОД И ВЫВОД

Программа не может считаться завершенной, полезной или интересной, если результат ее работы не может быть выведен и, если это требуется, в нее не могут быть введены некоторые данные. Технически язык программирования Си не имеет каких-либо средств для ввода или вывода. Однако средства ввода и вывода обеспечиваются посредством большого числа библиотечных функций. Использование этих функций настолько распространено, что их можно рассматривать как часть стандартной среды языка Си. Вполне возможно, что эти средства ввода и вывода будут включены в стандарт ANSI языка как часть самого языка Си.

В системе UNIX некоторое неотрицательное число, обозначающее файл, называется *дескриптором файла*. Целые числа 0, 1 и 2 являются дескрипторами файлов, которые обычно ассоциируются со стандартным файлом ввода, стандартным файлом вывода и стандартным файлом ошибок. Доступность этих дескрипторов файлов из любой программы на языке Си обеспечивается автоматически. Дескрипторы файлов предназначены для работы с файлами, с которыми они ассоциируются, а также

для работы с функциями ввода и вывода нижнего уровня, такими как `read` и `write`.

Большинство программ читают вводимые данные из специального файла - *потока* (*stream*). В поток записываются и выводимые данные. Поток - это файл, ассоциированный с буфером, позволяющим эффективный ввод и вывод на уровне пользователя.¹ В общем случае при вводе и выводе данных потоком прямой доступ к буферам потока невозможен; вместо него для управления потоками используются *потоковые указатели* (*stream pointer*). Потоковый указатель является ссылкой на структуру, содержащую информацию о соответствующем файле. Обычно для каждой программы автоматически открываются три потока, называемые стандартными потоками. К этим потокам можно обращаться с помощью постоянных указателей `stdin`, `stdout` и `stderr`, определения которых уже включены в файл стандартного пакета ввода-вывода `stdio.h`: `stdin` - для стандартного файла ввода, `stdout` - для стандартного файла вывода и `stderr` - для стандартного файла ошибок.

Потоки определяются с использованием заранее определенного типа `FILE` (в системе UNIX описание типа `FILE` дается в файле `stdio.h`). Стандартные потоки определяются как

```
FILE *stdin, *stdout, *stderr;
```

¹ Ввод из файла и вывод в файл наиболее эффективны, когда они выполняются блоками из *pbs* знаков, где *pbs* (*physical block size*) - обозначение размера физического блока, связанного с периферийным устройством, на котором хранится файл.

Если программа читает входные данные из файла или записывает выводимые данные в файл с использованием функций ввода и вывода нижнего уровня, таких как `read` и `write`, то для эффективной работы программы ввод и вывод должны выполняться блоками размером *pbs*. Если для ввода и вывода программой используются блоки других размеров, то для эффективной работы программы буферизация ввода и вывода должна быть организована самим программистом. До первого запроса программой вводимых данных *pbs* знаков читаются в массив; затем программа вместо чтения из файла читает данные из этого массива. Только когда массив становится пустым, выполняется следующее чтение из файла. Аналогично выводимые программой данные накапливаются в массиве; содержимое этого массива записывается в файл только при его заполнении или при завершении работы программы.

С другой стороны, если для ввода и вывода в программе используются потоки, то для программиста эта буферизация выполняется автоматически. В большинстве случаев более удобным оказывается использование для ввода и вывода потоков. Функции ввода и вывода нижнего уровня должны использоваться только в тех случаях, когда программист хочет управлять вводом и выводом на уровне детализации, недоступном с помощью потоков.

Чтобы открыть файл как поток и тем самым организовать буфер для этого файла, используется функция `fopen`.¹

Для чтения и записи в потоки стандартный пакет ввода-вывода (часть библиотеки `libc`, которая автоматически загружается с каждой программой компилятором языка Си) содержит множество функций. Ниже дано их краткое описание, дополнительные сведения об этих функциях и макросах приведены в приложении А.

Функции и макросы для ввода приводятся в следующей таблице:

| Функции и макросы ввода | Пояснения |
|----------------------------|--|
| <code>getc</code> | Извлекает следующий знак из указанного потока |
| <code>getchar</code> | Извлекает следующий знак из <code>stdin</code> |
| <code>fgetc</code> | Аналогична функции <code>getc</code> , но функция <code>fgetc</code> является исходной |
| <code>getw</code> | Извлекает следующее слово из указанного потока |
| <code>scanf</code> | Читает переменные различных типов из файла <code>stdin</code> , как указано в форматной строке |
| <code>fscanf</code> | Читает переменные различных типов из указанного потока, как указано в форматной строке |
| <code>gets</code> | Извлекает строку из файла <code>stdin</code> |
| <code>fgets</code> | Извлекает строку из указанного потока |

Функции и макросы для вывода приводятся в следующей таблице:

| Функции и макросы вывода | Пояснения |
|-----------------------------|--|
| <code>putc</code> (макрос) | Записывает знак в указанный поток |
| <code>putchar</code> | Записывает знак в файл <code>stdout</code> |

¹ Возможен и другой вариант открытия файла: сначала файл можно открыть, например, с помощью функции `open`, а затем преобразовать его в поток, используя функцию `fdopen`.

| | |
|----------------------|--|
| <code>fputc</code> | Аналогична функции <code>putc</code> , но функция <code>fputs</code> является исходной |
| <code>putw</code> | Записывает слово в указанный поток |
| <code>printf</code> | Записывает указанный список значений в файл <code>stdout</code> в соответствии с форматной строкой |
| <code>fprintf</code> | Записывает указанный список значений в указанный поток в соответствии с форматной строкой |
| <code>puts</code> | Записывает строку в файл <code>stdout</code> |
| <code>fputs</code> | Записывает строку в указанный поток |

Большинство из этих функций и макросов в качестве одного из своих аргументов воспринимают указатель потока. Однако некоторые из широко используемых функций и макросов ввода и вывода неявно используют файлы `stdin` и `stdout`. Ниже приведены несколько примеров, иллюстрирующих использование этих функций и макросов:

```

putchar(PROMPT);
printf("асору: невозможно открыть файл %s\n", src);
printf("%f\n", sine(x * 3.1416/180.0, eps));
/* sine - функция, определенная пользователем */
fprintf(stderr, "Ошибка - нет свободной памяти");
scanf("%f%c%f", &a,&opr,&b);
while ((c = fgetc(fs)) != EOF) ...;

```

Теперь о некоторых других функциях. Функция `ungetc` используется для возврата знака во входной поток. Функции `fseek` и `rewind` служат для изменения положения указателя файла, ассоциированного с потоком; указатель файла отмечает положение в потоке следующего элемента данных ввода или вывода. Функция `sprintf` используется для помещения выводимых данных в строку; с помощью функции `sscanf` можно прочитать ее входные данные из строки. Последние две функции можно применять для внутреннего преобразования элемента данных из одного формата в другой.

5.3.1. Использование потоков, определяемых программистом

С помощью функции `fopen` и соответствующего аргумента файл открывается для чтения или записи; функция `fopen` возвращает указатель на поток, ассоциированный с этим файлом. Чтение данных из файла можно выполнить функцией `fscanf`, а запись в файл - функцией `fprintf`. После обработки файл должен

быть закрыт вызовом функции `fclose`. После завершения работы программы все файлы закрываются автоматически.

Следующий программный сегмент иллюстрирует использование потоков, определяемых программистом:

```
FILE *fopen(), *fp;
.
.
.
/* "db_file": знаковый указатель, указывающий */
/* на строку, являющейся именем файла */
if ((fp = fopen(db_file, "r")) == NULL) {
    printf("Ошибка: нельзя открыть %s\n", db_file);
    exit(1);
}
.
.
.
while(fscanf(fp, "%s%s%s%s%s%s%s", dbfil->name,
    dbfil->room, dbfil->ext, dbfil->desig,
    dbfil->compid, dbfil->sig, dbfil->logid,
    dbfil->maild) != EOF)
{
    .
    .
    .
}
.
.
.
fclose(fp);
.
.
.
```

5.4. ПЕРЕНАЗНАЧЕНИЕ ВВОДА И ВЫВОДА (В СИСТЕМЕ UNIX)

Предположим, что некоторая программа на языке Си хранится в файле `enhance.c`, вводит данные из файла `stdin` и выводит их в файл `stdout`. Компиляция этой программы и подготовка ее выполнения в файле `enhance` являются результатом исполнения команды

```
cc -o enhance enhance.c
```

Программа `enhance` ожидает, что данные в нее будут вводиться прямо с терминала,¹ поскольку программа читает данные из файла `stdin`. Такой режим работы обеспечивается при вызове программы `enhance` как

```
enhance
```

По умолчанию, все данные, выводимые в поток ошибок `stderr`, посылаются на терминал.

Направление ввода для программы `enhance` можно изменить так, чтобы входные данные поступали из указанного файла, а не с терминала. Для этой цели служит оператор `<` переназначения ввода на командном уровне системы UNIX, например:

```
enhance <data
```

Выполнение этой команды обеспечивает получение входных данных программой `enhance` из файла `data`. (При этом программа `enhance.c` не изменяется!). Выходные данные по-прежнему посылаются на терминал.

Направление вывода из программы `enhance` также может быть изменено так, чтобы выводимые данные вместо терминала записывались в указанный файл. Это можно сделать, используя оператор `>` переназначения вывода на командном уровне системы UNIX, например:

```
enhance >result
```

Теперь данные при выводе из программы `enhance` посылаются в файл `result`. Данные, записываемые в стандартный поток ошибок, все еще посылаются на терминал. С помощью оператора `2>` можно переназначить и вывод данных об ошибках, например:

```
enhance >result 2>error
```

Можно также изменить направление одновременно для ввода и вывода, например:

```
enhance <data >result
```

¹ В системе UNIX конец ввода указывается вводом строки со знаком `control-D`, т.е. `^D`.

5.5. ГОЛОВНЫЕ ПРОГРАММЫ

Типичная головная программа имеет следующую форму:

```
операторы препроцессора - определения констант и макро,  
включение файлов  
внешние переменные  
main()  
{  
    определения и описания  
    операторы  
}  
функции
```

Все перечисленные компоненты программы будут содержаться в одном файле. Программу на языке Си можно также разделить на несколько частей так, что каждая часть будет содержаться в отдельном файле. Например, некоторые функции можно поместить в другой файл, допуская их независимую компиляцию. Если обращение к функции встречается раньше ее определения или ее определение находится в другом файле, необходимо наличие описания функции.

Первой выполняется функция с именем `main`.¹ Приведенная выше форма головных программ на языке Си описана неформально и не содержит всех возможных случаев. Например,

операторы препроцессора языка Си могут встречаться в любом месте файла, их расположение не ограничивается верхней частью программы; единственное ограничение состоит в том, что определения препроцессора должны предшествовать использованию определенных ими переменных.

Определению функции `main` могут предшествовать определения других функций.

Чтобы могли вызываться команды с параметрами, в языке Си допускается передача параметров в головную программу;² в следующем программном сегменте приводится формат описания параметров командной строки и доступа к ним:

¹ Такое соглашение используется в системе UNIX и в большинстве других операционных систем.

² Точнее говоря, специальное значение функции `main` и аргументов, которые могут быть ей переданы, является особенностью системы UNIX, а не языка Си. Язык Си так тесно связан с системой UNIX, что часто средство, рассматриваемое как часть языка Си, в действительности обеспечивается системой UNIX или ее командным языком.

```

main(argc, argv)
int argc;    /* число аргументов */
char *argv[]; /* массив "argv" содержит */
              /* указатели на аргументы */
              /* командного уровня      */
              /* системы UNIX */
{
    .
    .
    .
}

```

Предположим, что завершенная программа cmd.c прошла этап компиляции и под именем cmd вызвана на выполнение командой

cmd $a_1 a_2 \dots a_n$

В головной программе в файле cmd.c argc равен $n+1$, где n - число аргументов командной строки (argc равен $n+1$ потому, что имя команды неявно передается в качестве одного из этих аргументов). Массив argv содержит имя программы (cmd в приведенном выше примере) и имена аргументов. Элемент argv[0] - имя скомпилированной программы; элементы argv[1], argv[2], ..., argv[argc-1] являются аргументами команды argv[0]. Каждый из этих аргументов является строкой знаков, заканчивающейся нулевым знаком \0. И, наконец, argv[argc] есть 0, т.е. нулевой указатель.

В качестве примера рассмотрим команду echo, которая выводит аргументы своей командной строки:

```

/*-----*/
/* echo: печать аргументов командного уровня */
/*-----*/

```

```

#include <stdio.h>

```

```

main(argc, argv)
int argc;
char *argv[];
{
    int i;

    for (i = 1; i != argc; i++)
        printf("%s ", argv[i]);
    printf("\n");
}

```

```

    exit(0);
}
/*-----*/

```

5.6. ПРИМЕРЫ

В этом разделе будут даны несколько примеров с целью демонстрации универсальности языка Си при написании разнообразных программ. Первый пример демонстрирует использование языка Си для простых приложений в области обработки текстов. Второй пример, взятый из системы подготовки документов на ЭВМ, демонстрирует получение двумерного изображения бланка по его табличному описанию. В третьем примере текст, выводимый на дисплей типа HP2621 фирмы Hewlett-Packard, выделяется подчеркиванием, для чего необходимо посылать на дисплей управляющие символы, указывающие начало и конец подчеркивания. Четвертый и пятый примеры вычисления синуса и вычерчивания кривой по точкам служат иллюстрацией применения языка Си для научных задач. В шестом примере в процессе реализации алгоритма сортировки демонстрируется применение рекурсии. Седьмой пример показывает применение языка Си для написания простого драйвера устройства. В последнем примере при организации почти линейного поиска в массиве иллюстрируются элементарные приемы использования арифметических действий с указателями и описаний типа с применением оператора typedef и метки структуры.

5.6.1. Удаление последовательностей знаков форматирования из текста программы на языке Ада

Во время написания программы на языке Ада для публикации в книге [28] я должен был включить в ее текст знаки управления выбором шрифта, макросы и escape-знаки. Для тестирования программы прямо из ее машинно-читаемой формы знаки управления выбором шрифта, макросы и escape-последовательности необходимо было удалить из текста и заменить их, если необходимо, подходящими эквивалентами.

Нужно написать программу, которая выполняет следующие действия:

| Последовательность знаков | Пояснения | Действие |
|---------------------------|--------------------|----------|
| <code>\fI</code> | Перейти к курсиву | Удалить |
| <code>\fR</code> | Перейти к светлому | Удалить |

| | | |
|----------|---|----------------|
| \fP | прямому шрифту Вернуться к преды- дущему шрифту | Удалить |
| \fB | Перейти к жирному шрифту | Удалить |
| \- | Знак минус | Заменить на - |
| \$\$ | Знак минус | Заменить на - |
| \$\$\$ | Макро | Заменить на -- |
| *\$ | Макро | Заменить на * |
| \$star\$ | Макро | Заменить на * |
| \$app\$ | Макро | Заменить на ' |

В программе на языке Ада знаки \ и \$ используются только для указанных выше целей, ни для каких других целей они не употребляются.

В качестве примера рассмотрим следующую программу на языке Ада:

```

procedure SWAP(X, Y: in out FLOAT) is
  T:FLOAT; --temporary variable
begin
  T:=X;
  X:=Y;
  Y:=T;
end SWAP;

```

исходный текст которой имел следующий вид:

```

\fbprocedure\fr SWAP(X, Y: \fbin out\fr FLOAT) \fbis\fp
  T: FLOAT;  $$$temporary variable
\fbbegin\fr
  T := X;
  X := Y;
  Y := T;
\fbend\fr SWAP;

```

Прежде чем приведенный выше текст можно будет компилировать и выполнять как программу на языке Ада, из него надо удалить знаки форматирования, получив следующий текст:

```

procedure SWAP(X, Y: in out FLOAT) is
  T: FLOAT;  --temporary variable
begin
  T := X;
  X := T;

```

```

Y := T;
end SWAP;

```

Программа, удаляющая последовательности форматирующих знаков, базируется на следующем абстрактном алгоритме, который написан на смеси языка Си с русским языком:

```

while ((c = getchar()) != EOF) {
    if (c == '$') {
        c = getchar();
        switch (c) {
            case '-': заменить $-$ знаком минус; break;
            case 'd': заменить $dd$ двумя минусами; break;
            case 's': заменить $star$ знаком *; break;
            case '*': заменить $*$ знаком *; break;
            case 'a': заменить $app$ знаком ' ; break;
            default: напечатать сообщение об ошибке;
        }
    }
    else if (c == '\\') {
        c = getchar();
        switch (c) {
            case '-': заменить \- знаком минус; break;
            case 'f': удалить \fx, где x: B, I, R или P ; break;
            default: напечатать сообщение об ошибке;
        }
    }
    else
        putchar(c);
}

```

```

/*-----*/
/* Программа clean: удаление из программы на языке  */
/*                Ада знаков управления шрифтами  */
/*                и т.д.                             */
/*-----*/

```

```

#include <stdio.h>

main()
{
    int c;
    void replace();

    while ((c = getchar()) != EOF) {
        if (c == '$') {

```



```

c = getchar();
switch (c) {
case '-':
    replace("$", "-");
    break;
case 'd':
    replace("d$", "--");
    break;
case 's':
    replace("tar$", "*");
    break;
case '*':
    replace("$", "*");
    break;
case 'a':
    replace("pp$", "'");
    break;
default:
    fprintf(stderr, "clean: ошибка в применении $\n");
    exit(1);
}
}
else if (c == '\\') {
c = getchar();
switch (c) {
case '-':
    putchar('-');
    break;
case 'f':
    if ((c = getchar()) != 'B' && c != 'I' &&
        c != 'R' && c != 'P') {
        fprintf(stderr, "clean: ошибка, B, I,");
        fprintf(stderr, " Ожидаются знаки R или P\n");
        exit(1);
    }
    break;
default:
    fprintf(stderr, "clean: ошибка в применении \\ \n");
    exit(1);
}
}
else
    putchar(c);
}
}
/*-----*/
/*-----*/

```

```

/* replace(in, out): заменяет строку "in" на      */
/*                      строку "out"                */
/*-----*/

void replace(in, out)
char in[], out[];
{
    int i;

    for (i = 0; in[i] != NULL; i++) {
        if (getchar() != in[i]) {
            fprintf(stderr, "clean: ошибка, ожидается %c\n",
                        in[i]);
            exit(1);
        }
    }
    printf("%s", out);
}
/*-----*/

```

5.6.2. Получение изображения бланка из табличного описания

Напишем программу, которая, введя табличные данные, описывающие двумерный бланк, получает на экране дисплея двумерное изображение этого бланка. Для представления изображения имеется площадь размером 79 столбцов x 23 строки. Линии границы бланка проходят в столбцах 0 и 78 (помечаются знаком `)` и в строках 0 и 22 (помечаются знаком `-`).

Входные данные для программы в табличном виде задают координаты отображаемого элемента. Положение каждого элемента описывается строкой данных (одной на каждый элемент), имеющей для всех элементов один и тот же формат:

НомерСтроки НомерСтолбца ТекстОтображаемогоЭлемента

где *НомерСтроки* принимает значения между 1 и 21, *НомерСтолбца* - между 1 и 77. Например:

```

1 20 Анкета выпускника
3 5  Фамилия, инициалы
6 5  Название учебного заведения

```

и т.д. Текст программы приведен ниже:

```

/*-----*/
/* Программа main: преобразует табличное описание */

```

```

/*          бланка в его двумерное изображение*/
/*-----*/
#include <stdio.h>

#define NUMB_ROWS    23
#define NUMB_COLS    79
#define MAX_ROW (NUMB_ROWS-1)
#define MAX_COL (NUMB_COLS-1)

main( )
{
    int i,j; /* i - строка, j - столбец */
    char c;
    char display[NUMB_ROWS][NUMB_COLS];

    /* начальное заполнение площади изображения */
    /* знаками пробела и маркировка границы */
    for(i = 0; i <= MAX_ROW; i++)
        for(j = 0; j <= MAX_COL; j++)
            if (j == 0 || j == MAX_COL)
                display[i][j] = '|';
            else if (i == 0 || i == MAX_ROW)
                display[i][j] = '-';
            else
                display[i][j] = ' ';

    /* чтение табличного описания из стандартного */
    /* файла ввода и формирование бланка */
    while(scanf("%d%d%c", &i, &j, &c) != EOF)
        while ((c = getchar()) != '\n')
            display[i][j++] = c;

    /* вывод изображения бланка в файл стандартного вывода */
    for(i = 0; i <= MAX_ROW; i++) {
        for(j = 0; j <= MAX_COL; j++)
            putchar(display[i][j]);
        putchar('\n');
    }
    exit(0);
}

```

5.6.3. Улучшение изображения документа на терминале HP2621

Этот пример показывает, как программа пользователя может управлять форматом изображения. Несмотря на то, что рассматриваемый пример ориентирован на использование дис-

плея типа HP2621, программы для управления форматом изображения на других терминалах будут аналогичны программе, приведенной в этом подразделе.

В процессе подготовки электронного варианта бумажного документа важно отличать текст бланка от текста, подготовленного пользователем. На таких терминалах, как, например, растровые дисплеи, для знаков различных шрифтов можно выбрать различные изображения. На терминале HP2621 различных шрифтов нет, однако можно выделять знаки текста подчеркиванием. Для включения режима подчеркивания достаточно передать на терминал последовательность знаков ESC & d A, тогда последующие знаки при выводе на экран будут подчеркнуты; последовательность ESC & d @ выключает режим подчеркивания (в данном случае ESC обозначает escape-знак, которому соответствует код 27 стандарта ASCII).

Задача состоит в том, чтобы написать программу, получающую текстовый файл и формирующую новый файл с escape-последовательностями, которые обеспечивают подчеркивание букв и цифр при выводе модифицированного файла на экран дисплея.

Алгоритм программы получения файла с вышеупомянутыми последовательностями может быть описан как абстрактная программа:

```
while ((c = getchar()) != EOF)
    if (c - алфавитно-цифровой знак) {
        включить режим подчеркивания
        putchar(c);
        выводить все знаки до первого не алфавитно-цифрового знака
        выключить режим подчеркивания
        if (c != EOF) putchar(c);
    }
    else putchar(c);
```

Ниже приведен текст соответствующей программы на языке Си:

```
/*-----*/
/* Программа main: подчеркивание знаков для HP2621 */
/*-----*/

#include <stdio.h>
#include <ctype.h> /* содержит определение для */
                  /* функции "isalnum"; */
                  /* isalnum(c) возвращает */
                  /* значение "истина" (нену- */
```

```

        /* левое), если с - алфа- */
        /* витно-цифровой знак, и */
        /* значение "ложь" (нулевое)*/
        /* в противоположном случае */

#define START_UNDERLINE "\33&dA"
        /* 33 - десятичное 27; \33 представляет */
        /* escape-символ ESC */
#define STOP_UNDERLINE "\33&d@"

main()
{
    int c;

    while ((c = getchar()) != EOF)
        if (isalnum(c)) {
            /* включение функции подчеркивания */
            fputs(START_UNDERLINE, stdout);
            putchar(c);
            /* вывод всех знаков до первого */
            /* не алфавитно-цифрового знака */
            while (c = getchar(), isalnum(c))
                putchar(c);
            /* выключение функции подчеркивания */
            fputs(STOP_UNDERLINE, stdout);
            if (c != EOF)
                putchar(c);
        }
        else
            putchar(c);
    exit(0);
}

/*-----*/

```

Заметим, что программу, подобную рассмотренной, которая взаимодействует с оборудованием, невозможно написать на стандартном Паскале, так как управляющие знаки, такие как ESC, не являются допустимыми элементами заранее определенных типов языка Паскаль.

5.6.4. Функция *sine* вычисления значения синуса

Следующий пример иллюстрирует реализацию функции *sine* [91], точность результата которой (в пределах точности, достижимой с помощью действий с числами двойной точности) задает пользователь. Для практической работы более удобно, ко-

нечно, пользоваться функцией `sin` из математической библиотеки `libm` (см. приложение А).

Синус значения x (в радианах) вычисляется с помощью ряда

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots + (-1)^i \frac{x^{2i-1}}{(2i-1)!} + \dots$$

Вычисление синуса заканчивается, когда значение последнего члена ряда станет меньше или равно сумме предыдущих членов, умноженной на `eps` - произвольное малое значение.

Приведенный выше ряд состоит из рекурсивно определяемых членов

$$k_j = k_{j-1} + 2,$$

$$t_j = -t_{j-1} * x^2 / k_j(k_j - 1),$$

где $k_1 = 1$ и $t_1 = x$.

Следующая программа реализует функцию `sine`:

```

/*-----*/
/* Программа sine(x, eps): синус "x", вычисленный */
/* с точностью "eps" */
/*-----*/
#include <math.h> /* содержит функцию fabs, */
/* дающую абсолютное значение */
/* аргумента */
double sine(x, eps)
double x, eps;
{
    double sum, term;
    int k;
    term = x; k = 1; sum = term;
    while (fabs(term) > eps * fabs(sum))
    {
        k += 2;
        term *= (x * x) / (k * (k - 1));
        sum += term;
    }
    return sum;
}
/*-----*/

```

5.6.5. Метод наименьших квадратов для построения кривой [32]

Предположим, что имеется n измеренных значений, представляющих собой пары координат вида (x_i, y_i) , через которые

мы хотели бы провести прямые, задаваемые уравнением в форме

$$y(x) = a + bx,$$

где коэффициенты a и b являются параметрами. Значения этих параметров должны быть определены методом наименьших квадратов, который минимизирует сумму квадратов разностей между вычисленными и измеренными значениями, т.е. минимизирует значение функции

$$\sum_{i=1}^n [y(x_i) - y_i]^2$$

относительно коэффициентов a и b . Дифференцирование данной функции по этим параметрам после преобразований дает два уравнения

$$a + b \sum_{i=1}^n x_i = \sum_{i=1}^n y_i,$$

$$a \sum_{i=1}^n x_i + b \sum_{i=1}^n x_i^2 = \sum_{i=1}^n x_i y_i.$$

Из этих уравнений можно определить значения a и b :

$$b = \frac{\sum_{i=1}^n x_i \sum_{i=1}^n y_i - n \sum_{i=1}^n x_i y_i}{(\sum_{i=1}^n x_i)^2 - n \sum_{i=1}^n x_i^2},$$

$$a = \frac{\sum_{i=1}^n y_i - b \sum_{i=1}^n x_i}{n},$$

где $n \geq 2$ (поскольку для определения прямой необходимы по крайней мере две точки).

Приведенная ниже программа вводит измеренные значения координат и вычисляет коэффициенты a и b , используя рассмотренные выше уравнения.

```
/*-----*/
/* Программа main: аппроксимация кривой методом */
/*               наименьших квадратов           */
/*-----*/

#include <stdio.h>

#define MAX    500
```

```

main()
{
    float x[MAX], y[MAX]; /* измеренные значения */
    float a, b;           /* искомые коэффициенты */
                          /* уравнения "y(x) = a + bx" */

    int n;                /* число элементов */
    float p, q;           /* временные переменные */

    void least_square();

    n = 0;
    while (scanf("%f%f", &p, &q) != EOF) {
        if (n >= MAX) {
            printf("Допускается только %d значений\n", MAX);
            exit(1);
        }
        else {
            x[n] = p;
            y[n] = q;
            n++;
        }
    }

    if (n <= 1) {
        printf("Ошибка: менее 2 входных координат\n");
        exit(1);
    }

    least_square(x, y, n, &a, &b);

    printf("Коэффициенты a = %g и b = %g\n", a, b);

    exit(0);
}

/*-----*/
/*-----*/
/* Функция least_square: определяет значения */
/*                       параметров          */
/*-----*/

void least_square(x, y, n, pa, pb)
float x[], y[];
int n;
float *pa, *pb;
{

```



```

int i;
float sum_x = 0.0, sum_y = 0.0,
      sum_xy = 0.0, sum_x2 = 0.0;

for (i = 0; i < n; i++) {
    sum_x += x[i];
    sum_y += y[i];
    sum_xy += x[i] * y[i];
    sum_x2 += x[i] * x[i];
}
*pb = (sum_x * sum_y - n * sum_xy) /
      (sum_x * sum_x - n * sum_x2);
*pa = (sum_y - *pb * sum_x) / n;
}
/*-----*/

```

5.6.6. Быстрая сортировка

Напишем функцию для сортировки массива с элементами типа float, используя метод быстрой сортировки [35]. В этом алгоритме для быстрой сортировки применяется стратегия *разделяй и властвуй*. Сначала весь массив делится на две части, левую и правую, так, что все элементы левой части меньше или равны любому элементу правой части. Затем эти две части рекурсивно сортируются для получения упорядоченного массива.

Алгоритм быстрой сортировки более детально может быть описан с помощью следующего абстрактного алгоритма:

```

void quicksort(a, l, u);
float a[];      /* сортируемый массив */
int l, u;       /* границы массива */
{
    if (в массиве a меньше двух элементов)
        ничего не делать;
    else if (в массиве a содержится два элемента)
        упорядочить их (переставив при необходимости);
    else {
        разделить массив a на две части так, чтобы
            элементы левой части были меньше или равны r,
            элементы правой части были больше или равны r,
            (где r - произвольный элемент массива a)
        выполнить функцию quicksort для левой части массива;
        выполнить функцию quicksort для правой части массива;
    }
}

```

Чтобы гарантировать окончание рекурсии, необходимо для каждой части разбиения массива иметь хотя бы на один элемент меньше, чем в исходном массиве.

Разбиение массива на две части, как указано выше, может быть абстрактно описано следующим образом:¹

```
Пусть r - средний элемент массива a
i = l; j = u;
/* левая часть разбиения a[l..i-1] всегда содержит */
/* элементы <= r; в начальный момент оно пусто; */
/* аналогично правая часть разбиения a[j+1..u] */
/* всегда содержит элементы >= r */

while (i <= j) {
    Расширять левую часть разбиения a[l..i-1] увеличе-
    нием значения i, пока это возможно.
    Расширять правую часть разбиения a[j+1..u] уменьше-
    нием значения j, пока это возможно.
    Поменять местами элементы a[i] и a[j] так, чтобы
    могло продолжаться расширение разбиений массива a;
    изменить значения i и j
}
```

Может случиться, что при выполнении этого алгоритма две части разбиения пересекутся так, что в пересечении окажется элемент с значением, равным r. Такой элемент уже находится на своем месте - элементы слева от него меньше или равны r, а элементы справа - больше или равны r. По окончании цикла пересечение возможно при $i = j+2$ и невозможно при $i = j+1$. Избежать сортировки элемента пересечения можно выбором $a[l..j]$ в качестве левой части разбиения и $a[i..u]$ - в качестве правой.

На основе описанного выше алгоритма функция `partition` может быть определена следующим образом:

```
/*-----*/
/* partition: упорядочивает элементы массива типа */
/* float так, что элементы левой части */
/* разбиения <= элементов правой части */
/*-----*/
```

```
void partition(a, l, u, ri, rj)
```

¹ Обозначение $a[i..j]$ используется для ссылок на подмассив с элементами от $a[i]$ до $a[j]$. Если $i \leq j$, то рассматриваемый подмассив является нулевым массивом, т.е. массивом, не содержащим элементов.

```

float a[];    /* разбиваемый на части массив */
int l, u;     /* границы массива */
int *ri, *rj; /* искомые точки разбиения */
{
    float r, temp;
    int i, j;

    r = a[(l+u)/2]; /* средний элемент */
    i = l; j = u;

    while (i <= j) {
        /* расширить левую часть разбиения; */
        /* до завершения цикла при a[i] >= r */
        while (a[i] < r)
            i++;

        /* расширить правую часть разбиения; */
        /* до завершения цикла при a[j] <= r */
        while (a[j] > r)
            j--;

        /* поменять местами элементы i */
        /* изменить i и j */
        if (i <= j) {
            temp = a[i]; a[i] = a[j]; a[j] = temp;
            i++;
            j--;
        }
    }

    /* восстановить положение разбиения */
    *ri = i;
    *rj = j;
}

/*-----*/

```

Теперь может быть определена функция quicksort:

```

/*-----*/
/* quicksort: быстрая сортировка (эта программа */
/* сортирует только массивы типа float) */
/*-----*/

void quicksort(a, l, u)
float a[];
int l,u;

```

```

{
    int i, j;
    float temp;
    void partition();

    if (u-1 <= 0)
        ;
    else if (u-1 == 1) {
        if (a[u] < a[l]) {
            /* поменять местами a[l] и a[u] */
            temp = a[l]; a[l] = a[u]; a[u] = temp;
        }
    }
    else {
        partition(a, l, u, &i, &j);
        quicksort(a, l, j);
        quicksort(a, i, u);
    }
}

/*-----*/

```

5.6.7. Управление скоростью автомобиля

Напишем программу, обеспечивающую поддержание постоянной скорости движения автомобиля. Эта программа будет выполняться специализированным микропроцессором. Программа переходит в активное состояние, когда в ячейке памяти микропроцессора 020 появляется ненулевой код, и возвращается в пассивное состояние, когда код в этой ячейке становится нулевым. Текущее значение скорости автомобиля может быть получено из ячейки памяти 024. Изменить скорость можно записью некоторого значения в ячейку 026. После изменения содержимого этой ячейки необходимо примерно полсекунды, чтобы автомобиль отреагировал изменением скорости движения. Для задержки выполнения программы на период предписанного изменения скорости автомобиля используется функция delay, определенная вне рассматриваемой программы.

Допускается изменение скорости от 25 до 55 миль в час; при попытке драйвера установить скорость за указанными пределами в ячейку памяти 022 посылается ненулевой код, воспринимаемый как предупредительный сигнал. Такой же сигнал подается, если программа оказывается не в состоянии поддерживать заданную скорость с точностью до двух миль в час. В обоих случаях механизм управления должен отключаться автоматически.

Ниже приведен текст программы стабилизации скорости:

```

/*-----*/
/* Программа main: управление скоростью автомобиля */
/*-----*/

#define ABS(x)          (((x)>0)?(x):-(-x))
#define ON_OFF_ADDR     020
#define ALARM_ADDR      022
#define SPEED_ADDR      024
#define CHANGE_SPEED_ADDR 026

#define VARIATION        2
#define RESPONSE_TIME    0.5
#define LOW_LIMIT        25
#define HIGH_LIMIT       55
#define TRUE             1

main ()
{
    /* внешние связи */
    int *on = (int *) ON_OFF_ADDR;
    int *alarm = (int *) ALARM_ADDR;
    int *speed = (int *) SPEED_ADDR;
    int *change_speed = (int *) CHANGE_SPEED_ADDR;

    int current_speed, cruising_speed;
    void delay();

    for (;;) {      /* бесконечный цикл */

        while (*on == 0)
            ;

        cruising_speed = *speed;

        if (cruising_speed < LOW_LIMIT ||
            cruising_speed > HIGH_LIMIT) {
            *alarm = TRUE; *on = 0;
        }
        else {
            while (*on != 0) {
                current_speed = *speed;
                if (ABS(cruising_speed - current_speed)
                    > VARIATION) {
                    *alarm = TRUE;
                    *on = 0;
                    break;
                }
            }
            else

```

```

        *change_speed =
            cruising_speed - current_speed;

        delay(RESPONSE_TIME);
    }
}
}

/*-----*/

```

Напомним, что этой программе для выполнения требуется функция delay.

Программа управления скоростью автомобиля находится в состоянии ожидания до тех пор, пока не будет переведена в активное состояние. В состоянии ожидания постоянно выполняется проверка на появление какого-либо события вне программы. Такой режим работы приемлем только в некоторых ситуациях, например при выполнении программы на специализированном процессоре, и нежелателен при работе на компьютерах, на которых выполняется несколько задач в режиме разделения времени, так как приводит к нерациональному использованию системных ресурсов.

Вероятно, что в качестве микропроцессора для управления скоростью автомобиля используется микропроцессор с очень ограниченными возможностями. Программы для такого микропроцессора, написанные на языке высокого уровня, часто компилируются на более мощных компьютерах,¹ а результат трансляции переносится затем на микропроцессор. Такие программы часто сначала тестируются на тех же компьютерах, где они и компилируются в условиях, моделирующих реальные условия их работы. Приведенная выше программа тестировалась именно таким методом.

5.6.7.1. Реализация функции delay. Для реализации функции delay необходим доступ к часам реального времени, с помощью которых можно вычислить интервал задержки. Если же такой возможности нет, то приостановление процесса можно осуществить выполнением большого числа инструкций, например, таких, как в приведенном ниже цикле

```

for (i=0; i<=M; i++)
    for (j=0; j<=N; j++)
        ;

```

¹ Компиляция программы на одной машине для использования на другой называется кросс-компиляцией.

Время, необходимое для выполнения такого цикла, может быть определено аналитически исследованием кода, выданного компилятором, с использованием таблицы времени выполнения команд микропроцессора или экспериментально - выполнением программы на микропроцессоре. Необходимой задержки можно добиться подбором значений параметров M и N .

5.6.8. Передача функций в качестве параметров

Напишем функцию `integrate` для нахождения определенного интеграла вещественной функции f

$$I = \int_{i=a}^{i=b} f(x) dx$$

с использованием правила трапеций. При реализации функции `integrate` необходимо обеспечить возможность интегрирования различных функций с помощью передачи интегрируемой функции в качестве параметра функции `integrate`.

В соответствии с правилом трапеций интеграл функции в пределах от a до b аппроксимируется площадью трапеции с основанием $b - a$ и высотами $f(a)$ и $f(b)$.

Точность аппроксимации можно повысить, если интервал от a до b разделить на два подынтервала, вычислить площади двух полученных трапеций и сложить их. Процесс такого деления может быть продолжен. При использовании n подынтервалов интеграл I_n вычисляется по формуле

$$I_n = h(f(a)/2 + f(a+h) + f(a+2h) + \dots + f(a+(n-1)h) + f(b)/2),$$

где ширина интервала h определяется как $h = (b-a)/n$. Процесс деления продолжается до тех пор, пока разность двух последовательных аппроксимаций интеграла функции f , вычисленных с помощью правила трапеций, не будет отличаться на значение, меньшее, чем заданное значение ϵ ($\epsilon > 0$).

Абстрактное описание соответствующего алгоритма интегрирования может иметь следующий вид:

```
new_approx = 0;
do {
    previous_approx = new_approx;
    Вычислить новое значение аппроксимации new_approx;
    Разделить каждый интервал на два равных подынтервала;
} (abs(new_approx - previous_approx) >= eps);
```

На основании этого алгоритма функция `integrate` определяется как

```

/*-----*/
/* integrate: в соответствии с точностью и пределами */
/*             интегрирования, заданными пользователем, */
/*             вычисляется определенный интеграл */
/*             вещественной функции (заданной как */
/*             "float") по правилу трапеций */
/*-----*/

#include <math.h> /* для функции abs (абсолютное */
                  /* значение) используются опре- */
                  /* деления библиотеки math */

float integrate(fp, a, b, eps)
float (*fp)(), a, b, eps;
    /* "fp" - указатель на интегрируемую функцию; */
    /* "a" и "b" - пределы интегрирования; */
    /* "eps" - требуемая точность */
{
    float new_approx = 0.0; /* начальное значение */
    float previous_approx;
    int n = 1; /* число интервалов */
    double h; /* длина интервала */
    double sum; /* временная переменная */
    int i; /* переменная цикла */

    h = (b - a); /* начальная длина */
                /* интервала */

    do {
        previous_approx = new_approx;
        /* вычислить новое значение аппроксимации */
        sum = (*fp)(a) / 2.0;
        for (i = 1; i < n; i++)
            sum += (*fp)(a + i * h);
        sum += (*fp)(b) / 2.0;
        new_approx = sum * h;
        n *= 2; /* число интервалов для следующей */
                /* аппроксимации */
        h /= 2.0; /* длина интервала для следую- */
                 /* щей аппроксимации */
    } while (fabs(new_approx - previous_approx) >= eps);
    return new_approx;
}
/*-----*/

```

Вместо использования библиотечной функции `fabs` для вычисления абсолютного значения пользователь может опреде-

лечь собственную функцию или макрос, например взять макрос ABS из программы управления скоростью автомобиля:

```
#define ABS(x) (((x)>0)?(x):- (x))
```

Если первая аппроксимация при вычислении интеграла функции f при $\text{eps}=0.0$ дает значение, совпадающее с начальным значением, присвоенным переменной `new_approx`, то функция `integrate` дает неверный результат. Эта трудность преодолевается заданием по меньшей мере двух аппроксимаций.

Вызов функции `integrate` очень прост. Например, для функции `cube`, описанной как

```
float cube();  
/* возвращает куб ее аргумента */
```

значение интеграла от 1 до u с точностью `small` можно получить следующим вызовом функции `integrate`:

```
integrate(cube, 1, u, small)
```

Функция `integrate` служит также примером использования различных составных операторов присваивания.

5.6.9. Поиск в массиве

Таблица `pt` (см. определение ниже) некоторого процесса представляет собой массив, элементы которого являются структурами типа `pcb`:

```
typedef struct pcb  
{  
    long pid;  
    proc_states state;  
    long delay;  
    int fp;  
    struct pcb *parent;  
} pcb;
```

Обратите внимание на комбинацию оператора `typedef` и метки структуры. Идентификатор `pcb` описан одновременно как тип и как метка структуры. Необходимость в описании `pcb` как метки структуры следует из рекурсивности описания идентификатора `pcb`; с другой стороны, хотя нет необходимости описывать идентификатор `pcb` с помощью оператора `typedef` как имя типа, такое описание позволяет использовать идентификатор `pcb` аналогично заранее определенным типам (т.е. при описании или опре-

делении структур типа `pcb` тогда не нужно использовать ключевое слово `struct`). Используемый в описании `pcb` перечисляемый тип `proc_states` описывается как

```
typedef enum
{
    ready,
    service,
    select,
    transaction,
    completed
} proc_states;    /* состояния процесса */
```

Эти описания типов, а также описания констант и переменных, содержатся в файле `proc.h`:

```
#define NULL            0
#define NULL_PID        (-1)

extern pcb pt[];        /* таблица процесса */
extern pcb *pt_last;    /* последний элемент */
                        /* таблицы процесса */
```

Внешние переменные определяются и инициализируются в другом файле. Адрес первого элемента таблицы `pt` обозначается просто `pt` (или `&pt[0]`), адрес последнего элемента задается указателем `pt_last`.

Осталось написать функцию `next_ready`, возвращающую адрес (а не индекс) элемента таблицы `pt`, компонент которого `pid` имеет значение, отличное от `NULL_PID`, а значение компонента `state` равно `ready`. Если такой элемент не найден, функция `next_ready` должна возвращать значение `NULL`.

```
/*-----*/
/* next_ready: находит адрес элемента таблицы */
/*           процесса с значением ready */
/*-----*/

#include "proc.h"

pcb *next_ready()
{
    pcb *p;

    for (p = &pt[0]; p < pt_last; p++) {
        if (p->pid != NULL_PID && p->state == ready)
            return p;
    }
    return NULL;
}
```

```

    return NULL;
}
/*-----*/

```

Выполнение выражения `p++` приводит к увеличению указателя `p` на значение `sizeof(pcb)`, а не на единицу.

5.7. ЗАДАЧИ

1. Напишите функцию `sqrt` для вычисления квадратного корня из значения `x` типа `float`, используя метод Ньютона. В соответствии с этим методом $(k+1)$ -я аппроксимация квадратного корня из значения `x` определяется по формуле

$$a_{k+1} = 0.5(a_k + x/a_k).$$

Итеративный процесс заканчивается, когда абсолютное значение разности между двумя последовательными аппроксимациями становится меньше заданного пользователем значения `eps`.

2. Программа аппроксимации кривой методом наименьших квадратов рассчитана на четное число входных значений. Хотя ввод нечетного числа значений приводит к бессмысленному результату, программа не воспринимает такой случай как ошибочный. Измените программу так, чтобы при попытке ввода нечетного числа значений выдавалось сообщение об ошибке. (Подсказка: функция `scanf` возвращает число успешно обработанных элементов, если же вводимые данные заканчиваются раньше, чем будет считан первый элемент данных, то возвращается признак EOF.)

3. Напишите функцию `merge` слияния двух массивов `a` и `b` для получения третьего отсортированного массива `c`:

```
void merge(a, b, c).
```

4. Напишите функцию `mergesort` сортировки массива по методу *разделяй и властвуй* аналогично тому, как он был использован в функции `quicksort`. Основная идея алгоритма заключается в разбиении массива на две части, в сортировке каждой части с последующим слиянием обеих частей (используя функцию `merge` предыдущей задачи) для получения отсортированного массива. Абстрактное описание алгоритма для функции `mergesort` может иметь следующий вид [66]:

```

void mergesort(a, l, u);
float a[];    /* сортируемый массив */
int l, u;     /* границы массива */
{

```

```

if (число элементов a больше или равно 2) {
    Разделить a на две непустые части p и q;
    Выполнить функцию mergesort разбиения p;
    Выполнить функцию mergesort разбиения q;
    Выполнить функцию mergesort(p, q, a);
}
}

```

5. Ряды для вычисления функции $\sin x$ быстро сходятся только для малых значений x , удовлетворяющих условию

$$0 \leq \text{abs}(x) \leq \pi/4$$

Для больших значений x нужно использовать следующие тождества [91]:

Тождество

Условия

$$\sin x \equiv \sin(x - 2\pi n)$$

$$2\pi n \leq \text{abs}(x) < 2\pi(n+1)$$

$$\sin x \equiv -\sin(x - \pi)$$

$$\pi \leq \text{abs}(x) < 2\pi$$

$$\sin x \equiv \sin(\pi - x)$$

$$\pi/2 \leq \text{abs}(x) < \pi$$

$$\sin x \equiv \cos(\pi/2 - x)$$

$$\pi/4 < \text{abs}(x) < \pi/2$$

$$\sin x \equiv -\sin(-x)$$

$$x < 0$$

где $\cos x$ задается рядом

$$\cos(x) = 1 - x^2/2! + x^4/4! - \dots$$

Измените функцию `sine`, используя эти тождества.

6. В примере, в котором используется метод наименьших квадратов для аппроксимации кривой, программа не будет надежно работать, так как она не проверяет четность числа введенных пользователем координат. Введите такую проверку в программу.

7. В системе UNIX знаки `^H` (backspace) и `@` часто используются в качестве специальных знаков, которые стирают соответственно последний введенный знак и удаляют последнюю строку. Напишите программу, которая считывает текст, содержащий эти знаки, и выдает отредактированный этими знаками вариант исходного текста.

Измените приведенную в этой главе программу `echo` так, чтобы ее аргументы печатались с новой строки, если в качестве первого аргумента программы указывается `-N`. Например:

```
echo a b c
```

выводится на печать как

a b c

a строка

```
echo -N a b c
```

вызывает печать аргументов в виде

```
a  
b  
c
```

Аргумент командной строки в форме *-буква* (такой, как *-N*) называется в соответствии с терминологией системы UNIX флагом или опцией команды.

Глава 6

Раздельная компиляция и абстрагирование данных

Текст программы, написанной на языке Си, может целиком содержаться в одном файле,¹ а может быть распределен по нескольким файлам. Файлы, содержащие компоненты программ (функции, описания и определения), могут компилироваться независимо. Компоненты программ, прошедшие раздельную компиляцию, и функции из библиотеки заранее откомпилированных функций могут быть объединены в одну законченную программу.

Поскольку файлы с текстами на языке Си могут компилироваться раздельно, большую программу целесообразно разделить на небольшие части, работать с которыми более удобно. Программа может быть разделена на группы логически связанных компонентов, таких как константы, переменные, типы и функции. Разбиение программ на меньшие части облегчает построение, понимание и сопровождение больших систем [40]. Раздельная компиляция позволяет отдельно для каждого файла выполнить проверку на синтаксические и семантические ошибки и даже провести тестирование на ошибки времени выполнения. Более того, при изменении программы необходимо выполнить повторную компиляцию только измененных компонентов (для автоматизации этого процесса часто используется про-

¹ Файлом называется компонента, поддерживаемая операционной системой компьютера и служащая для постоянного хранения текста или данных.

грамма make; более детально - см. приложение Б). Перекомпиляция после каждого изменения в большой программе нежелательна из-за существенных затрат времени и ресурсов компьютера.

Файлы могут служить также в качестве средства сокрытия информации и инкапсуляции данных. К объектам других файлов можно обращаться только в том случае, если эти объекты являются внешними с классом памяти `extern`; к объектам, в описании которых указан класс памяти `static`, обращение из других файлов невозможно. Таким образом, с помощью файлов можно управлять видимостью объектов. Например, программист может указать, к каким из функций, определенным в некотором файле, допускается обращение из других файлов, а к каким - нет. Такие детали реализации функций, как структуры используемых или разделяемых ими данных, можно сделать недоступными для пользователя. Следовательно, его программа, использующая эти функции, не будет зависимой от особенностей их реализации. Таким образом, после достижения соглашения о спецификациях всех видимых компонентов файла способы реализации или проведения изменений в этих компонентах могут быть любыми, если сохраняется соответствие согласованным спецификациям.

В файле с текстом на языке Си могут содержаться определения констант, определения и описания объектов и определения функций; для таких файлов предлагается следующий формат:

*определения констант
внешние определения и описания
определения функций*

Синтаксис внешних определений и описаний точно такой же, как и определений и описаний внутри функций с одним лишь отличием - тела функций могут быть только во внешних определениях (функции не могут быть вложенными).

6.1. ОБЛАСТЬ ДЕЙСТВИЯ ВНЕШНИХ ОПРЕДЕЛЕНИЙ И ОПИСАНИЙ

Существует два вида областей действия, ассоциированных с внешними идентификаторами: *лексическая область действия* и *внешняя область действия* (или видимость). Лексическая область действия была определена ранее как область программы, где действует ее описание или определение.

Внешняя область действия определяется как часть программы, где все ссылки на один и тот же идентификатор являются обращениями к одному и тому же объекту. Внешние объекты и функции могут иметь класс памяти `extern` (который является также классом памяти, присваиваемым по умолчанию) или

static Область действия внешнего описания или определения с классом памяти `static` ограничивается файлом, содержащим эти определения и описания. Область действия других внешних описаний и определений (с классом памяти `extern`) включает в себя все другие файлы, составляющие программу.

Как уже упоминалось, наличие ключевого слова `extern` указывает, что память, ассоциированная с этим описываемым идентификатором, будет выделена в другом файле. С внешними описаниями функций дело обстоит несколько иначе: все внешние описания функций эквивалентны независимо от наличия ключевого слова `extern`; например, описания

```
extern void sort();  
void sort();
```

эквивалентны. В отличие от описания функции, определение функции перечисляет формальные параметры и их описания и определяет тело функции. Естественно, для каждой функции программы должно быть только одно внешнее определение (т.е. определение без ключевого слова `static`).

6.2. РАЗДЕЛЬНАЯ КОМПИЛЯЦИЯ

Законченная программа на языке Си (содержащая функцию `main`, если речь идет о системе UNIX), части которой находятся в различных файлах, скажем в `file1.c`, `file2.c`, ..., `filen.c`, может быть скомпилирована в одну выполняемую программу, скажем `final`, следующим образом:¹

```
cc -o final file1.c file2.c ... filen.c
```

Можно также выполнить компиляцию каждого файла отдельно; по окончании компиляции всех файлов их скомпилированные версии могут быть объединены в законченную программу. Например, предположим, что файлы `filei.c` были скомпилированы по отдельности с использованием команд

```
cc -c filei.c
```

¹ По соглашению, в системе UNIX имена файлов с программами на языке Си имеют суффикс `.c`. Этому же соглашению следуют и другие средства для работы с программами на языке Си, например такие, как программа проверки типов `lint` и программа сопровождения группы программ `make`.

и полученным в результате компиляции *объектными*¹ файлам присвоены имена *file.o*.² Тогда для связывания этих файлов, т.е. их объединения в выполняемую версию программы, можно дать команду

```
cc -o final file_1.o file_2.o ... file_n.o
```

6.3. АБСТРАКТНЫЕ ТИПЫ ДАННЫХ И СОКРЫТИЕ ИНФОРМАЦИИ

Для реализации абстрагирования данных файлы могут использоваться точно так же, как и подпрограммы, используемые для абстрагирования управления и сокрытия информации. Абстрактным объектом данных называется такой объект, с которым допускаются только те операции, которые разрешены при его определении. Прямой доступ к кодам, реализующим абстрактный объект данных, закрыт для пользователя, скрыты от него также все детали реализации абстрактного объекта данных. Сокрытие этих деталей не дает пользователю возможности:

1. Разрабатывать программы, зависящие от представления абстрактных типов данных, чтобы изменение этого представления не влияло на остальную часть программы. (Например, начальной реализацией абстрактного типа данных *set* может быть массив, затем это представление может быть заменено на упорядоченный список для более эффективного использования памяти.)

2. Случайно или умышленно нарушить целостность объектов абстрактного типа данных. (Целостность объектов абстрактного типа данных сохраняется благодаря тому, что пользователь вынужден выполнять только те операции, которые предусмотрены создателем этого абстрактного типа данных.)

В качестве примеров абстрактных объектов данных можно указать очереди, множества, базы данных и двоичные деревья.

6.3.1.Стек - пример абстрактного типа данных

Рассмотрим абстрактный объект данных, с которым допускается выполнение только следующих операций:

| | |
|----------------------|--|
| <code>push(i)</code> | добавить целый элемент <i>i</i> к абстрактному объекту данных; |
|----------------------|--|

¹ Объектным файлом называется результат трансляции на машинный язык (т.е. файл, полученный компилятором) программы, написанной на языке высокого уровня.

² По соглашению, имена объектных файлов, полученных с помощью компилятора языка Си в системе UNIX, имеют суффикс *.o*.

| | |
|---------|---|
| pop() | удалить верхний элемент из абстрактного объекта данных; |
| top() | дать значение верхнего элемента (самого последнего добавленного элемента) абстрактного объекта данных; |
| empty() | вернуть значение TRUE, если абстрактный объект данных пуст; в противном случае вернуть значение FALSE; |
| full() | вернуть значение TRUE, если абстрактный объект данных полон; в противном случае вернуть значение FALSE; |
| clear() | очистить абстрактный объект данных. |

Объект данных, с которым допускается выполнение перечисленных выше операций, называется стеком.

С помощью приведенных ниже определений и описаний реализуется стек целых элементов. Все эти определения и описания должны содержаться в одном файле. В реализации стека используется целый массив *s*, в котором хранятся заносимые в стек значения. Переменная *next* служит для указания следующего свободного элемента в массиве *s*. Ограничением видимости массива *s* и переменной *next* пределами файла, их содержащего, скрыты детали реализации от тех, кто будет этим стеком пользоваться; для ограничения видимости используется класс памяти *static*.

```

/*-----*/
/* push, pop, top, empty, full, clear: функции для */
/*                               выполнения */
/*                               операций со стеком */
/*-----*/

#define MAX_SIZE    200
static int s[MAX_SIZE];
static int next = 0;
    /* указывают на первый свободный элемент стека */

/*-----*/

void push(i)        /* добавляет элемент "i" в стек */
int i;
{
    if (next == MAX_SIZE) {
        printf("push: Ошибка: стек полон\n");
        exit(1);      /* завершение программы */
    }
    s[next++] = i;
}

```

```

/*-----*/
void pop()      /* удалить элемент из вершины стека */
{
    if (next == 0) {
        printf("pop: ошибка: стек пуст\n");
        exit(1);
    }
    next--;
}
/*-----*/
int top() /* возвращает значение элемента из      */
        /* вершины стека */
{
    if (next == 0) {
        printf("top: ошибка: стек пуст\n");
        exit(1);
    }
    return s[next-1];
}
/*-----*/
int empty() /* возвращает значение TRUE, если      */
           /* стек пуст, и значение FALSE - в     */
           /* противном случае                    */
{
    return next == 0;
}
/*-----*/
int full() /* возвращает значение TRUE, если      */
          /* стек полон, и значение FALSE в      */
          /* противном случае                    */
{
    return next == MAX_SIZE;
}
/*-----*/
void clear() /* очищает стек */
{
    next = 0;
}
/*-----*/

```

Из других файлов можно обращаться только к этим функциям, поскольку они имеют класс памяти `extern` (присвоенный по умолчанию).

Попытки втолкнуть элемент в стек, когда он полон, либо попытки удаления элемента или определения значения элемента в вершине стека, когда стек пуст, приводят к завершению программы.

6.3.2. Ограниченность файлов как средств абстрагирования данных

Файл нельзя рассматривать как средство для решения всех задач абстрагирования данных. Например, невозможно определить массив файлов или определить указатель на файл. В частности, в примере со стеком невозможно прямо определить массив стеков или передать стек как параметр.

Введение файлов не ставило своей целью создать механизм сокрытия информации и абстрагирования данных! Файлы предназначались в качестве средства обеспечения разбиения программ на части и раздельной компиляции.

6.4. КЛАССЫ

Для развития средств структурирования данных к языку Си было добавлено новое средство абстрагирования данных - *класс* (*class*) [82 - 84], базирующееся на аналогичном средстве языка Симула 67.

В настоящее время классы не являются частью стандарта языка Си [77] и, таким образом, не поддерживаются стандартными компиляторами языка; классы включены только в язык Си++, являющийся расширением языка Си, и поддерживаются компилятором Си++.¹ Однако возможно, что в недалеком будущем классы все же будут включены в стандарт языка Си.

Здесь классы рассматриваются по следующим причинам:

чтобы читатель освоился со средствами абстрагирования данных;

для иллюстрации дополнительных возможностей, обеспечиваемых использованием классов, по сравнению с оператором `typedef`;

чтобы показать язык Си в его развитии.

Как уже указывалось, тип - это множество значений и операций с этими значениями. Применение классов позволяет вводить полноправные типы, определенные пользователем, в то время как оператор `typedef` в действительности не является средством описания типов, поскольку он не обеспечивает механизма для описания операций. Он позволяет только описывать множество значений, а операции, ассоциированные с описываемым типом данных, определены быть не могут.

Описание класса состоит из двух частей: спецификации класса и определения функций, представленных в описании класса.

¹ Язык Си++ является стандартным языком Си, в который добавлены новые средства, такие как классы, расширение операторов и функций, внутренние функции, новый синтаксис описания функций и проверка аргументов функций.

Спецификации класса аналогичны описаниям структур. Они имеют следующую форму:

```
class имя-класса
{
    закрытые описания
public:
    открытые описания
};
```

Открытые и закрытые описания состоят из описаний данных и функций. Описания функций несколько отличаются от описаний функций стандартного языка Си - в интересах улучшения проверки типов и удобства чтения программы должны быть также указаны типы формальных параметров.

Пользователь имеет доступ только к объектам, описанным в открытой части описания класса. Объекты, описания которых даны в закрытой части описания класса, пользователю недоступны. Доступ к ним возможен только в определениях функций, данных в описании класса.

Открытая часть класса является интерфейсом между пользователем и абстрактным типом данных, реализованным классом; закрытая часть класса содержит детали реализации, сведения о которых не предназначены для пользователя. Доступ к классу можно регулировать, помещая соответствующие объекты в закрытую или открытую часть класса. Такое регулирование дает уверенность, что программа, использующая этот класс, остается независимой от представления класса, и что сохраняется целостность объектов класса.

Для доступа к компонентам объекта класса используются те же обозначения, что и для доступа к элементам структуры, т.е. *объект-класса* . *элемент-класса*. Определения функций даются отдельно. Например, функция *f*, определяемая в спецификации класса *c*, должна быть определена отдельно как

```
type c.f(описания параметров)
{
    .
    .
    .
}
```

Синтаксически функция класса немного отличается от обычной функции. Имя функции класса должно содержать имя класса в качестве префикса, а описания формальных параметров должны быть даны в заголовке функции. Рассмотрим описание класса

stack для реализации целых стеков. Его спецификация имеет следующий вид:

```
#define MAX_SIZE    200

class stack {
    int s[MAX_SIZE];
    int next;
public:
    void push(int); /* указывается тип формального */
                  /* параметра */
    void pop();
    int top();
    int empty();
    int full();
    void clear();
};
```

Функции, описанные в классе, определяются теперь следующим образом:

```
/*-----*/

void stack.push(int i)    /* обратите внимание на */
                        /* описание формального */
                        /* параметра */
{
    if (next == MAX_SIZE) {

        printf("push: ошибка: стек полон\n");
        exit(1);    /* завершение программы */
    }
    s[next++] = i;
}

/*-----*/

void stack.pop()
{
    if (next == 0) {
        printf("pop: ошибка: стек пуст\n");
        exit(1);
    }
    next--;
}

/*-----*/
```

```

int stack.top()
{
    if (next == 0) {
        printf("top: ошибка: стек пуст\n");
        exit(1);
    }
    return s[next-1];
}
/*-----*/
int stack.empty()
{
    return next == 0;
}
/*-----*/
int stack.full()
{
    return next == MAX_SIZE;
}
/*-----*/
void stack.clear()
{
    next = 0;
}
/*-----*/

```

Исходные тексты спецификации класса и функций класса обычно помещаются в разные файлы.

Классы могут использоваться аналогично типам в описаниях объектов, например:

```
stack s[5], sa, sb, *ps;
```

Переменная `s` представляет собой массив стеков, переменные `sa` и `sb` являются отдельными стеками, а переменная `ps` - указателем на стек. Работа с этими стеками допускается только с помощью функций, описанных в классе `stack`:

```

s[3].clear(); /* инициализация стека "s[3]" */
sa.push(i+14); /* добавление элемента "i+14" */
               /* к стеку "sa" */

```

Область памяти для объектов класса может выделяться или освобождаться с помощью заранее определенных операторов `new` и `delete`:

```

ps = new stack;
delete ps;

```

Использование классов имеет важное преимущество в том, что изменение представления класса и его закрытых описаний не влияет на программу, использующую этот класс. Например, в реализации класса `stack` массив может быть заменен списком, на использование этого класса такое изменение не окажет никакого влияния.

Важное различие в использовании классов и файлов для реализации абстрагирования данных состоит в том, что классы являются полноправными типами, а файлы - нет. Например, используя стек, определенный с помощью класса, можно определить массив стеков или указатель на стек, что невозможно в случае с файлами.

6.4.1. Конструкторы, деструкторы и расширение операторов и функций в языке Си++

В языке Си++ имеется ряд других средств, связанных с понятием класса; эти средства перечислены ниже.

6.4.1.1. *Конструкторы*. Одна из функций конструктора, связанная с типом класса, выполняется автоматически при определении объекта класса. Например, эта функция может использоваться для очистки стеков во время их определения.

6.4.1.2. *Деструкторы*. Функция деструктора, связанная с типом класса, автоматически выполняется при выходе за пределы области действия объекта класса. Такие функции могут использоваться для очистки объекта перед его уничтожением, например, с их помощью можно явным указанием освободить память, ранее занимаемую объектами, чтобы ее можно было использовать для других объектов.

6.4.1.3. *Расширение операторов и функций*. Операторы и функции могут расширяться, т.е. один и тот же символ операции или идентификатор функции могут быть ассоциированы с более чем одной операцией или функцией. В частности, такая возможность оказывается полезной при использовании классов. Например,

1. Расширение дает пользователю возможность распространить действие обычных арифметических операторов на комплексные числа (для использования совместно с классом типа `(complex)`), определенным пользователем, обеспечивая, таким образом, естественные обозначения для действий с комплексными числами).

2. Расширение оператора присваивания позволяет осуществить присваивание объектов класса (расширение оператора присваивания позволяет неявно передавать объекты класса в качестве параметров).

6.4.2. Заключительные замечания о классах

Введение в язык Си классов существенно расширило возможности структурирования программ. Выше были показаны лишь простейшие примеры использования классов; например, не рассматривался вывод новых классов из существующих или использование существующих классов для построения новых классов [82]. Читателю, желающему получить дополнительные сведения о классах (и о языке Си++), рекомендуется обратиться к соответствующей литературе [82-84].

6.5. ПРИМЕРЫ

В этом разделе приведены три примера реализации абстрагирования данных. В первых двух примерах задача решается с помощью файлов, в третьем - с помощью классов.

6.5.1. Таблица символов

Напишем функции для реализации таблицы символов. Для удобства использования таблицы символов необходимо также предоставить пользователю файл, содержащий набор описаний. С помощью функций, предназначенных для работы с таблицей символов, выполняются следующие действия:

1. Занести в таблицу символов элемент вместе с соответствующей информацией (элементами являются строки).
2. Получить из таблицы символов информацию об элементе.
3. Определить, заполнена ли таблица символов.
4. Проверить наличие символа в таблице символов.
5. Восстановить начальное состояние таблицы символов.

Записать элемент в таблицу символов можно только в том случае, если таблица не заполнена. В таблице символов должно быть достаточно места для 200 элементов. С каждым элементом в таблицу символов записывается информация, указывающая, является ли элемент идентификатором, именем процедуры или функции, ключевым словом или именем метки. Приведенный ниже пример является упрощенным вариантом реальной таблицы символов.

Описания типов, предназначенные для функций таблицы символов и ее пользователей, содержатся в файле `syntab.h`:

```
/*-----*/  
/* syntab.h: файл описаний для таблицы символов      */  
/*-----*/  
  
typedef enum {var, fun, proc, key_word,  
              label} item_type;
```



```

typedef struct {
    char *id;
    item_type t;
} item_info;

void add(), clear();
int in_table(), full();
item_type get();

/*-----*/

```

Теперь можно определить функции таблицы символов следующим образом:

```

/*-----*/
/* add, in_table, full, get, clear: функции таблицы */
/*                                     символов */
/*-----*/

#include "syntab.h"

#define N 200
#define TRUE 1
#define FALSE 0

int strcmp(); /* функция сравнения строк из */
               /* стандартной библиотеки */
char *strcpy(); /* функция копирования строк из */
                /* стандартной библиотеки */

static item_info st[N]; /* таблица символов */
static int next = 0; /* точки входа в таблицу: */
                    /* st[0..next-1] */

/*-----*/

void add(x, it)
char *x;
item_type it;
{
    char *malloc(); /* функция выделения памяти из */
                   /* автоматически загружаемой */
                   /* стандартной библиотеки "libc" */
    int strlen(); /* функция получения длины строки */
                 /* из стандартной библиотеки */

    if (next == N) {

```

```

    printf("add: переполнение таблицы символов\n");
    exit(1);
}

/* выделить память в таблице символов для элемента */
/* "x" и скопировать туда "x" */
st[next].id = malloc((unsigned) strlen(x) + 1);
strcpy(st[next].id, x);
st[next].t = it;
next++;
}

/*-----*/

int in_table(x)
char *x;
{
    int i;

    for(i = 0; i < next; i++)
        if (strcmp(x, st[i].id) == 0)
            return TRUE;
    return FALSE;
}

/*-----*/

item_type get(x) /* прежде чем вызывать функцию */
                /* "get", необходимо удостовериться, */
                /* что элемент "x" находится в */
                /* таблице символов */
char *x;
{
    int i;

    for(i = 0; i < next; i++)
        if (strcmp(x, st[i].id) == 0)
            return st[i].t;
    printf("get: элемента %s нет в таблице символов\n", x);
    exit(1);
}

/*-----*/

int full()
{
    return next == N;
}

```

```

/*-----*/

void clear()
{
    next = 0;
}

/*-----*/

```

При отсутствии доступной памяти функция `malloc` возвращает нулевое значение указателя `NULL` (0). Обращение к нулевой ячейке памяти приводит к появлению невразумительного сообщения об ошибке вида

```
Bus error - core dumped
```

Об этой ошибке можно сообщить более понятно, если добавить фрагмент программы

```

if (st[last].id == NULL) {
    printf("add: ошибка: нет свободной памяти\n");
    exit(1);
}

```

после вызова функции `malloc`:

```
st[last].id = malloc((unsigned) strlen(x)+1);
```

Запрос на выделение памяти и выдачу сообщения об ошибке можно записать вместе:

```

if ((st[last].id=malloc((unsigned) strlen(x)+1))
    == NULL) {
    printf("add: ошибка: нет свободной памяти\n");
    exit(1);
}

```

Приведенная выше простая организация таблицы символов оказывается неэффективной при большом числе запросов на поиск. В такой ситуации для поиска лучше использовать хеширование или упорядоченные двоичные деревья.

6.5.2. Функции для обработки списков

Функции для обработки списков неформально могут быть описаны следующим образом:

| | |
|------------------|---|
| add(head, i) | Добавить целое i к списку, указываемому значением head |
| delete(head, i) | Удалить целое i из списка |
| in_list(head, i) | Вернуть значение TRUE, если целое i содержится в списке, иначе вернуть значение FALSE |
| empty(head) | Вернуть значение TRUE, если указываемый значением head список пуст, и значение FALSE в противном случае |

В отличие от функций для работы с таблицей символов, которые были предназначены для работы только с одной таблицей, функции для обработки списков получают указатель на заголовок списка в качестве параметра и поэтому могут работать с различными списками. Заголовки списков могут быть описаны в программе пользователя как указатели на элементы списков, имеющие тип node. Описание типа node, как и другие описания, даны в приведенном ниже файле list.h:

```

/*-----*/
/* list.h: файл описаний для целых функций обработки */
/*          списков                                     */
/*-----*/

struct node {
    int value;
    struct node *next;
};

void add(), delete();
int in_list(), empty();

/*-----*/

```

Файл list.h включается в файл с определениями функций, он должен также включаться в файлы с обращениями к этим функциям.

Ниже приведены тексты функций для обработки списков:

```

/*-----*/
/* add, delete, in_list, empty: целые функции для */
/*          обработки списков                     */
/*-----*/

#include <stdio.h>
#include "list.h"

```

```

#define TRUE 1
#define FALSE 0

void add(phead, i)    /* необходимо передать адрес */
                    /* списка, т.е. указатель на */
                    /* заголовок списка */
struct node **phead; /* обратите внимание на двой- */
                    /* ную косвенную ссылку */
int i;
{
    struct node *t;
    char *malloc(); /* функция выделения памяти */

    /* поиск в списке значения "i" */
    t = *phead;
    while (t != NULL && t->value != i)
        t = t->next;

    if (t == NULL) { /* вставить "i" в список, (так */
                    /* как его нет в списке) */
                    /* выделить память для одного элемента */
                    t = (struct node *)
                        malloc(sizeof(struct node));
                    /* присвоить этому элементу значение "i" */
                    t->value = i;
                    /* занести элемент в заголовок списка */
                    t->next = *phead; phead = t;
    }
}

/*-----*/

void delete(phead, i)
struct node **phead;
int i;
{
    struct node *t, *temp;
    int free();

    if (*phead == NULL)
        ; /* пустой список */
    else {
        if ((*phead->value == i) {
            /* нужно удалить первый элемент списка */
            temp = *phead;
            *phead = (*phead->next;
            free((char *) temp);
        }
    }
}

```

```

    else {
        t = *phead;
        while (t->next != NULL && t->next->value != i)
            t = t->next;
        if (t->next != NULL) {
            temp = t->next;
            t->next = t->next->next;
            free((char *) temp);
        }
    }
}

/*-----*/

int in_list(phead, i)
struct node **phead;
int i;
{
    struct node *t;

    for (t = *phead; t != NULL; t = t->next)
        if (t->value == i)
            return TRUE;
    return FALSE;
}

/*-----*/

int empty(phead)
struct node **phead;
{
    return *phead == NULL;
}

/*-----*/

```

6.5.3. Класс buffer

Предположим, что поставлена задача реализовать кольцевой буфер размером 128 байтов для работы со знаковыми элементами. Для этой цели опишем класс `buffer`. Для работы с объектами класса `buffer` необходимо обеспечить следующие операции:

| | |
|----------------------|--|
| <code>empty()</code> | Возврат значения TRUE, если буфер пуст; в противном случае возврат значения FALSE. |
| <code>clear()</code> | Очистка буфера; эта же функция может быть использована для инициализации буфера. |
| <code>put(c)</code> | Добавление знака с в буфер, если он не полон; при успешном завершении возвращается значение 0; в противном случае возвращается значение -1, которое сигнализирует об ошибке. |
| <code>get()</code> | Возвращение следующего знака, полученного из буфера, если буфер не пуст; если же буфер пуст, то возвращается значение -1 в качестве индикатора ошибки. |

Описание класса `buffer` имеет следующий вид:

```
define SIZE    128

class buffer {
    int s[SIZE];
    int in, out, count;
public:
    int empty();
    void clear();
    int put(char);
    int get();
};
```

Ниже приведены тексты функций класса `buffer`:

```
#define TRUE    1
#define FALSE   0
#define FAILURE (-1)
#define SUCCESS 0

/*-----*/

int buffer.empty()
{
    return count == 0;
    /* можно было бы написать вместо этого */
    /* оператора просто "return !count"? */
    /* почему? */
}
```

```

/*-----*/

void buffer.clear()
{
    in = out = count = 0;
}

/*-----*/

int buffer.put(char c)
{
    if (count == size)
        return FAILURE;
    else {
        s[in] = c;
        in = (in + 1) % SIZE; /* вернуться на начало */
                               /* при достижении конца */
                               /* буфера */
        count++;
        return SUCCESS;
    }
}

/*-----*/

int buffer.get()
{
    char result;

    if (count == 0)
        return FAILURE;
    else {
        result = s[out];
        out = (out + 1) % SIZE; /* вернуться на начало */
        count--;
        return result;
    }
}

/*-----*/

```

6.6. ЗАДАЧИ

1. Измените реализацию стека (ту, в которой для абстрагирования данных используются файлы) так, чтобы функции стека не завершали программу при попытке добавить элемент в заполненный стек, удалить элемент из пустого стека и т. д.

Сделайте так, чтобы в указанных ситуациях функции возвращали значение -1, а при нормальном завершении - значение 0.

2. В реализации функции стека `push` вместо сравнения значений `pext` и `MAX_SIZE` для определения, заполнен стек или нет, можно было бы использовать функцию `full`. Аналогично в реализации функции `pop` можно было бы использовать функцию `empty`. Проанализируйте достоинства и недостатки обоих подходов.

3. В примере с таблицей символов функция `clear` очищает таблицу присваиванием переменной `last` значения 0. При этом память, которую занимали элементы таблицы `st`, не может использоваться повторно (т.е. будет потеряна), поскольку память, занимаемая таблицей символов, не освобождается явным указанием (например, с помощью функции `free`). Измените функцию `clear` так, чтобы перед присваиванием переменной `last` значения 0 освобождалась память, занимаемая элементами таблицы `st`.

4. С чем связано введение ограничения на число элементов в таблице символов? Можно ли снять ограничение в 200 элементах, используя для реализации таблицы символов список вместо массива? Попробуйте это сделать. При реализации удостоверьтесь, что проведенные изменения незаметны для пользователя, т.е. синтаксически и семантически действие функций не изменилось с точки зрения пользователя.

5. Измените пример с классом `stack`, чтобы для хранения элементов стека можно было использовать списки. Проанализируйте все "за" и "против" использования массивов и списков для хранения элементов.

6. Что нужно сделать, чтобы в стеке можно было хранить как целые числа, так и числа с плавающей точкой? Подсказка: воспользуйтесь объединениями (`union`).

Глава 7

Особые ситуации

Особой ситуацией называется такое событие, которое происходит неожиданно или редко, как, например, деление на ноль или преждевременное прерывание выполнения программы. Так же, как в языках Паскаль, Фортран или Алгол, но в отличие от языков ПЛ/1 и Ада, в языке Си отсутствуют специальные средства для обработки особых ситуаций.

В языках, не имеющих средств для обнаружения особых ситуаций и их обработки, для этих целей должны применяться другие методы. Наиболее общий подход состоит во введении для индикации особых состояний *кодов состояния* [61]. Если зна-

чение, возвращенное функцией, указывает, что во время ее выполнения возникла особая ситуация, то предпринимаются соответствующие действия. Однако применение кодов состояния еще не решает задачи обнаружения и обработки особых состояний процессора, таких как попытка деления на ноль; то же можно сказать про особые ситуации, причиной возникновения которых является среда программы, например особая ситуация, вызванная желанием пользователя прервать выполнение программы. Техника, основанная на применении кодов состояния, должна быть дополнена некоторыми другими средствами, которые, возможно, обеспечиваются операционной системой.

Для обработки особых ситуаций в языке Си используются следующие средства:

1. *Коды состояния.* По соглашению, функции возвращают значение -1, если в процессе их выполнения возникла особая ситуация. Например, при обнаружении конца файла функция `getc` возвращает значение -1.

2. *Сигналы в системе UNIX.*¹ При получении сигнала формируется обращение к операционной системе для вызова специальных функций. Чтобы воспользоваться такими средствами обработки особых состояний, в программу на языке Си нужно включить библиотечные функции, взаимодействующие с операционной системой. Сигналы генерируются программно или аппаратно.

Сигналы в системе UNIX в общем случае используются для обработки [61]:

1. Особых ситуаций, возникающих под воздействием среды, влияющей на выполнение программы (например, вводом знака удаления генерируется сигнал, указывающий на необходимость завершения выполняемой в данный момент программы).

2. Особых ситуаций, обнаруживаемых аппаратурой (например, запрещенное обращение к памяти).

3. Особых ситуаций, которые можно было бы обработать возвращением кодов состояния.

Сигналы формируются автоматически при ошибках в программах или явно посылаются одним процессом другому.²

Использование кодов состояния для индикации особых ситуаций является очевидным решением и было проиллюстри-

¹ Механизм сигналов в системе UNIX является асинхронным механизмом взаимодействия между процессами. Сигналы используются также для индикации появления особых состояний. В соответствии с терминологией системы UNIX слова *сигнал* и *особое состояние* будут далее считаться взаимозаменяемыми, если не будет возникать сомнений относительно их смысла.

² Пользователь, вводящий знак удаления для прерывания выполняемого в настоящий момент процесса, вызывает тем самым генерацию соответствующего сигнала программой, управляющей клавиатурой.

ровано многими примерами в предыдущих главах. В этой главе основное внимание будет уделено обработке особых ситуаций с помощью сигналов системы UNIX. Однако следует предупредить, что перенос программ, использующих сигналы, в другую операционную систему может потребовать некоторых изменений.

7.1. РАЗЛИЧНЫЕ СИГНАЛЫ

В программах на языке Си, выполняемых в системе UNIX, может обрабатываться множество различных сигналов. Их описания содержатся в файле `<signal.h>`. Некоторые из обрабатываемых в системе UNIX [3] сигналов приведены ниже:

| Номер | Имя | Комментарий |
|-------|---------|--|
| 1 | SIGHUP | Зависание программы |
| 2 | SIGINT | Прерывание |
| 3 | SIGQUIT | Выход из программы без сохранения данных |
| 4 | SIGILL | Запрещенная инструкция |
| 5 | SIGTRAP | Захват |
| 6 | SIGIOT | Инструкция обращения к супервизору при вводе-выводе |
| 7 | SIGEMT | Инструкция обращения к супервизору |
| 8 | SIGFPE | Особая ситуация при выполнении операции с плавающей точкой |
| 9 | SIGKILL | Прекращение работы |
| 10 | SIGBUS | Ошибка шины |
| 11 | SIGSEGV | Нарушение сегментации |
| 12 | SIGSYS | Ошибочный аргумент в системном вызове |
| 13 | SIGPIPE | Запись для межпроцессорного обмена при отсутствии процесса, читающего данные |
| 14 | SIGALRM | Предупреждение в заданное время |
| 15 | SIGTERM | Программный сигнал завершения |
| 16 | SIGUSR1 | Определенный пользователем сигнал 1 |
| 17 | SIGUSR2 | Определенный пользователем сигнал 2 |
| 18 | SIGCLD | Уничтожение подчиненного процесса |
| 19 | SIGPWR | Отказ источника питания |

Обработка особых ситуаций, таких как арифметическое переполнение или деление на ноль, зависит от реализации. Например, большинство реализаций языка Си игнорирует переполнение при обработке целых чисел [77].

7.2. УСТАНОВКА РЕЖИМА ОБРАБОТКИ ОСОБЫХ СИТУАЦИЙ

Для установки режима обработки сигналов, т.е. обработки особых ситуаций, вызывается функция `signal` системы UNIX. Ее спецификация имеет следующий вид:

```
#include <sys/signal.h>

int (*signal(sig, func))()
int sig;
int (*func)();
```

Функция `signal` имеет два формальных параметра: `sig`, который определяет обрабатываемый сигнал, и `func`, который является указателем на функцию, определяющую способ обработки этого сигнала; эта функция вызывается при получении сигнала `sig`, а ее предыдущее значение возвращается функцией `signal` (в случае ошибки возвращается значение -1).

Фактический параметр, соответствующий формальному параметру `func`, может иметь одно из трех возможных значений: `SIG_DEL`, `SIG_IGN` или *указатель на функцию*. Этими значениями предписываются следующие действия:¹

| Значение <code>func</code> | Действие |
|----------------------------|---|
| <code>SIG_DEL</code> | Завершить процесс при получении сигнала |
| <code>SIG_IGN</code> | Игнорировать сигнал |
| <i>адрес функции</i> | При получении сигнала выполнить указанную функцию |

Следующий сегмент программы иллюстрирует установление соответствия между функцией *handler* обработки сигнала и сигналом `signal_name`:

```
#include <signal.h>    /* описание сигнала */
{
    .
    .
    .
```

¹ Компонент программы, который выполняется параллельно с остальной частью программы, называется *процессом*. Подробно процессы обсуждаются в гл. 8.

```

    /* связать сигнал с обрабатывающей его функцией */
    signal(signal_name, handler);
    .
    .
    .
}

```

Другой пример, приведенный ниже, связывает функцию обработки сигнала с сигналом только в том случае, если пользователь ранее не указывал, что сигнал должен игнорироваться; необходимость в использовании этого примера возникает только в контексте, в котором пользователь мог указать, что сигнал должен игнорироваться:

```

#include <signal.h>
{
    int (*old)();
    .
    .
    .
    if ((old = signal(signal_name, SIG_IGN)) != SIG_IGN)
        signal(signal_name, handler);
    .
    .
    .
}

```

Ссылка на исходную функцию обработки сигнала запоминается в переменной *old*, чтобы позднее можно было восстановить ее связь с сигналом *signal_name*. Если нет необходимости запоминать исходную функцию обработки сигнала, то приведенный выше оператор *if* может быть записан как

```

if (signal(signal_name, SIG_IGN) != SIG_IGN)
    signal(signal_name, handler);

```

Функция обработки сигнала *handler* должна иметь следующую форму:

```

int handler(i) /* обработка сигнала "i" */
int i;
{
    .
    .
    .
    /* восстановить связь сигнала "i" с обрабатывающей */
}

```

```

/* его функцией */
signal(i, handler);
.
.
.
}

```

Когда получен сигнал, для которого указана обрабатывающая особую ситуацию функция, нормальное выполнение программы приостанавливается и управление передается этой функции; сигнал ей передается в качестве аргумента. При нормальном завершении функции обработки особой ситуации выполнение программы возобновляется с той точки, где она была прервана. В некоторых случаях выполнение программы не может быть продолжено, например:

функция обработки особой ситуации может завершить выполнение программы вызовом функции `exit`;
продолжение программы может возобновиться не с точки прерывания, если функция обработки особой ситуации вызовет функцию `longjmp` (не локальный `goto`).

В функции обработки особой ситуации необходимо восстанавливать связь сигнала и функции, так как при каждом следующем появлении сигнала в большинстве случаев, по умолчанию, выполняются заранее предусмотренные действия.¹

7.2.1. Пример использования сигналов для обработки ошибок при выполнении операций с плавающей точкой

Программа, которую предстоит написать, должна вычислять вещественные корни квадратного уравнения; если же выполнение программы прекращено вследствие возникновения особого состояния при вычислениях с плавающей точкой (таких как деление на нуль, переполнение или исчезновение порядка), то должно быть выдано соответствующее сообщение об ошибке. Особые ситуации при вычислениях с плавающей точкой генерируют сигнал `SIGFPE`.

Два корня r_1 и r_2 квадратного уравнения

$$ax^2 + bx + c = 0.0$$

определяются из соотношений

¹ В новой версии системы Berkeley UNIX (т.е. версии 4.2) нет необходимости восстанавливать связь сигнала и обрабатывающей функции, поскольку во время обработки сигнала связь с обрабатывающей функцией не устанавливается.

$$r_1 = \frac{-b - \sqrt{b^2 - 4.0ac}}{2.0a}$$

и

$$r_2 = \frac{-b + \sqrt{b^2 - 4.0ac}}{2.0a}.$$

Эти корни будут действительными только при

$$b^2 - 4.0ac > 0.0.$$

Программа вычисления корней приведена ниже:

```

/*-----*/
/* main: программа вычисления вещественных корней */
/*    квадратного уравнения */
/*-----*/

#include <stdio.h>
#include <signal.h> /* в системе UNIX 5.0 исполь- */
                  /* зуются "<sys/signal.h>" */
#include <math.h>   /* содержит описание для */
                  /* функции "sqrt" */

main()
{
    float a, b, c;
    float r1, r2;
    float temp;

    int float_error();

    /* установка функции обработки особых ситуаций */
    signal(SIGFPE, float_error);

    printf("Введите 3 коэффициента, разделенные пробелами:");
    if (scanf("%f %f %f", &a, &b, &c) != 3) {
        printf("Ошибка: не хватает входных данных\n");
        exit(1);
    }

    if (a == 0.0) {
        printf(" Уравнение не квадратное,");
        printf(" так как первый коэффициент равен 0\n");
    }

```

```

    exit(1);
}

if (b * b - 4.0 * a * c < 0.0) {
    printf("Уравнение не имеет вещественных корней\n");
    exit(1);
}

temp = (float) sqrt(b * b - 4.0 * a * c);

r1 = (- b + temp) / (2.0 * a);

r2 = (- b - temp) / (2.0 * a);

printf("Корни уравнения: %g и %g\n", r1, r2);
}
/*-----*/

/*-----*/
/* float_error: функция обработки особых ситуаций */
/*               при вычислениях с плавающей точкой */
/*-----*/

int float_error(i)
int i;
{
    printf("Особая ситуация! \n");
    exit(1);
}
/*-----*/

```

Функция обработки особых ситуаций с плавающей точкой `float_error` не может напечатать переменные `a`, `b` и `c`, поскольку не имеет к ним доступа; эти переменные являются локальными для функции `main`. Однако если бы они были глобальными для функции `main`, например внешними переменными, то их значения можно было бы напечатать из функции `float_error` и тем самым лучше проинформировать пользователя о причине ошибки.

Если бы в этой программе не было вызова функции обработки особых ситуаций, то при возникновении такой ситуации программа была бы прервана с малопонятным сообщением об ошибке.

Вышеприведенная программа использует функцию `sqrt`, содержащуюся в библиотеке математических функций `libm`; следовательно, она должна компилироваться с этой библиотекой. Поскольку библиотека математических функций уже скомпилиро-

вана, лучше было бы функцию `sqrt` включить в программу на этапе редактирования связей (предполагая, что программа вычисления корней содержится в файле `roots.c`), например:

```
cc -o roots roots.c -lm
```

7.3. ГЕНЕРАЦИЯ И ПОСЫЛКА СИГНАЛОВ

Сигналы могут генерироваться явно или неявно. Сигналы генерируются в программе неявно соответствующим программным или аппаратным обеспечением при возникновении каких-либо необычных ситуаций, например при делении на ноль с плавающей точкой, переполнении в операциях с плавающей точкой или при вводе пользователем знака "удалить" с терминала.

Сигналы генерируются явно, когда один процесс посылает сигнал другому процессу или самому себе вызовом функции `kill`, которая имеет следующую спецификацию:

```
int kill(pid, sig)
int pid, sig;
```

Аргумент `pid` является номером идентификации процесса, которому посылается сигнал `sig`.

7.4. ПРИМЕРЫ

Приведенная в гл. 1 в качестве примера программа калькулятора будет здесь модифицирована так, чтобы в ней могли обрабатываться особые ситуации с помощью сигналов, генерируемых как явно, так и неявно. Для иллюстрации семантики обработки особых ситуаций будут представлены три варианта программы.

7.4.1. Программа калькулятора с обработкой особых ситуаций (вариант 1)

Модифицируем программу калькулятора, приведенную в гл. 1, так, чтобы в ней можно было обрабатывать особые ситуации при вычислениях с плавающей точкой и сигналы преждевременного завершения программы; сигнал преждевременного завершения выполняемому в данный момент процессу может быть послан вводом знака "удалить" (`delete character`) с терминала. В том случае, если программе послан сигнал завершения, необходимо получить подтверждение от пользователя; если пользователь отвечает утвердительно, программа должна быть завершена; в противном случае выполнение программы должно быть продолжено.

Модифицированная программа калькулятора имеет следующий вид:

```
/*-----*/
/* main: простой калькулятор с обработкой особых */
/*        ситуаций (вариант 1)                    */
/*-----*/

#include <stdio.h>
#include <signal.h> /* в системе UNIX 5.0 исполь- */
                   /* зуются файл "sys/signal.h" */

#define PROMPT ':'

main()
{
    int float_error(), term_inter();
    float a, b;
    char opr;
    float result;

    /* установить функции обработки прерываний и */
    /* обработки особых ситуаций */
    signal(SIGFPE, float_error);
    signal(SIGINT, term_inter);

    while(putchar(PROMPT),scanf("%f%c%f",&a,&opr,&b)!=EOF)

        switch (opr) {
            case '+': result = a + b; break;
            case '-': result = a - b; break;
            case '*': result = a / b; break;
            default:
                printf("Ошибка ***, неверен знак операции\n");
                exit(1);
        }

        printf("Результат равен %g\n",result);
    }
    exit(0);
}

/*-----*/
/* float_error: функция обработки особых ситуаций */
/*              при выполнении операций с плавающей */
/*              точкой                               */
/*-----*/
```

```

int float_error(i)
int i; /* обрабатываемый сигнал */
{
    signal(SIGFPE, float_error);

    printf("Ошибка при выполнении операции с плавающей\n");
    printf("точкой; результат операции неверен\n");
}
/*-----*/

/*-----*/
/* term_inter: функция обработки прерываний, зада- */
/* ваемых пользователем с клавиатуры, */
/* например, знаками "del" или "break" */
/*-----*/

int term_inter(i)
int i; /* обрабатываемый сигнал */
{
    int c;

    signal(SIGINT, term_inter);

    printf("Вы действительно хотите закончить? Д или Н:");

    c = getchar();
    switch (c) {
        case 'Д': case 'д': exit(0);
        default: printf("продолжение работы\n");
    }
}
/*-----*/

```

7.4.2. Программа калькулятора с обработкой особых ситуаций (вариант 2)

В предыдущем варианте программы калькулятора ошибка при выполнении операции с плавающей точкой может привести к заиклииванию, поскольку на некоторых компьютерах после возврата из функции обработки особых ситуаций ошибочная операция будет выполняться снова.

Проблема заиклиивания может быть решена, если после возврата из функции обработки особых ситуаций обеспечить условия, при которых ошибка не повторится. Простой способ решения этой задачи состоит в изменении значений операндов оши-

бочной операции при выполнении функции обработки особых ситуаций. В этом примере источником возникновения особых ситуаций может быть одна из операций +, -, * или /; чтобы гарантировать отсутствие особой ситуации после возврата из функции обработки особых ситуаций `float_error`, достаточно операциям `a` и `b` перечисленных выше операций присвоить значения, равные 1.0. Чтобы функция `float_error` могла изменить их значения, переменные `a` и `b` должны быть глобальными, т.е. внешними.¹

```

/*-----*/
/* main: Простой калькулятор с обработкой особых */
/*       ситуаций (вариант 2)                      */
/*-----*/

#include <stdio.h>
#include <signal.h> /* в системе UNIX 5.0 исполь-   */
                  /* зуются файл "sys/signal.h" */

#define PROMPT ':'

float a, b;
char opr;
float result;

main()
{
    int float_error(), term_inter();

    /* установить функции обработки прерываний и */
    /* обработки особых ситуаций */
    signal(SIGFPE, float_error);
    signal(SIGINT, term_inter);

```

¹ Предупреждение: такое решение зависит от реализации языка, поскольку оно опирается на предположение, что после возврата из функции обработки особых ситуаций `float_error` при возобновлении нормального выполнения программы будут использоваться новые значения переменных `a` и `b`. Будут эти новые значения использоваться или нет, зависит от способа генерации кода компилятором. Например, если значение переменной `b` хранится в регистре, то его изменение в функции обработки особых ситуаций не обязательно приведет к использованию нового значения этой переменной при возобновлении нормальной работы программы.

Компилятор языка Си системы Berkeley UNIX [5] использует новые значения `a` и `b`, а компилятор Си системы AT&T UNIX [3] принимает другое решение. Для последнего компилятора должен применяться метод с использованием функции `longjmp`; этот метод лучше, так как он не зависит от реализации языка.

```

while(putchar(PROMPT),scanf("%f%c%f",&a,&opr,&b)!=EOF)

switch (opr) {
    case '+': result = a + b; break;
    case '-': result = a - b; break;
    case '*': result = a / b; break;
    default:
        printf("Ошибка *** неверен знак операции\n");
        exit(1);
}

printf("Результат равен %g\n",result);
}
exit(0);
}

/*-----*/
/* float_error: функция обработки особых ситуаций */
/*              при выполнении операций с плавающей */
/*              точкой */
/*-----*/

int float_error(i)
int i; /* обрабатываемый сигнал */
{
    signal(SIGFPE, float_error);

    printf("Ошибка при выполнении операции с плавающей\n");
    printf("точкой; результат операции неверен\n");

    /* изменить операнды для предотвращения особых */
    /* ситуаций в операциях с плавающей точкой */
    a = b = 1.0
}
/*-----*/

/*-----*/
/* term_inter: функция обработки прерываний, зада- */
/*              ваемых пользователем с клавиатуры, */
/*              например, знаками "del" или "break" */
/*-----*/

int term_inter(i)
int i; /* обрабатываемый сигнал */
{
    int c;

    signal(SIGINT, term_inter);

```

```

printf("Вы действительно хотите закончить? Д или Н:");

c = getchar();
switch (c) {
case 'Д': case 'д': exit(0);
default: printf("Продолжение работы\n");
}

}
/*-----*/

```

7.4.3. Программа калькулятора с обработкой особых ситуаций (вариант 3)

Хотя последняя модификация программы решает проблемы с использованием функции `float error`, остаются некоторые трудности с использованием второй функции обработки особых ситуаций `term_inter`. Предположим, что пользователь прерывает программу вводом знака `del` в тот момент, когда программа читает данные, и затем дает отрицательный ответ на вопрос о выходе из программы. После возврата из функции обработки особых ситуаций функция `scanf` вернет значение EOF, которое вызовет завершение программы. Завершение программы можно предотвратить, с помощью функции `longjmp` сохранив среду в той точке, где должно возобновиться выполнение программы, а затем использовать функцию `longjmp` (не локальный оператор `goto`) в функции обработки особых ситуаций для возобновления работы программы; вызов `longjmp(env, 0)` восстанавливает среду, сохраненную в буфере `env` последним вызовом функции `longjmp`.

```

/*-----*/
/* main: простой калькулятор с обработкой особых */
/* ситуаций (вариант 3) */
/*-----*/

#include <stdio.h>
#include <signal.h> /* в системе UNIX 5.0 исполь- */
                  /* зуются файл "sys/signal.h" */
#include <setjmp.h>

#define PROMPT ':'

float a, b;
char opr;
float result;

```

```

jmp_buf env; /* буфер для сохранения состояния */
              /* программы, которое необходимо при */
              /* восстановлении выполнения программы */
              /* после возврата из терминальной */
              /* функции обработки особых ситуаций */
              /* "term_inter" */

main()
{
    int float_error(), term_inter();

    /* установить функции обработки прерываний и */
    /* обработки особых ситуаций */
    signal(SIGFPE, float_error);
    signal(SIGINT, term_inter);

    /* сохранить начальное состояние области */
    setjmp(env);

    while(putchar(PROMPT),scanf("%f%c%f",&a,&opr,&b)!=EOF)

        switch (opr) {
            case '+': result = a + b; break;
            case '-': result = a - b; break;
            case '/': result = a / b; break;
            default:
                printf("Ошибка *** неверен знак операции\n");
                exit(1);
        }

        printf("Результат равен %g\n",result);
    }
    exit(0);
}

/*-----*/
/* float_error: функция обработки особых ситуаций */
/*              при выполнении операций с плавающей */
/*              точкой */
/*-----*/

int float_error(i)
int i; /* обрабатываемый сигнал */
{
    signal(SIGFPE, float_error);

    printf("Ошибка при выполнении операции с плавающей\n");

```

```

printf("точкой; результат операции неверен\n");

/* изменить операнды для предотвращения особых */
/* ситуаций в операциях с плавающей точкой */
a = b = 1.0
}
/*-----*/

/*-----*/
/* term_inter: функция обработки прерываний, зада- */
/*           ваемых пользователем с клавиатуры, */
/*           например, знаками "del" или "break" */
/*-----*/

int term_inter(i)
int i; /* обрабатываемый сигнал */
{
    int c;
    void longjmp();

    signal(SIGINT, term_inter);

    printf("Вы действительно хотите закончить? Д или Н:");

    c = getchar();
    switch (c) {
        case 'Д': case 'д': exit(0);
        default: printf("продолжение работы\n");
    }
}
/*-----*/

```

7.5. ЗАДАЧИ

1. Что случится, если в программе калькулятора пользователь в ответ на вопрос

Вы действительно хотите закончить? Д или Н:

заданный функцией `term_inter`, вместо "Д" или "Н" введет знак "удалить"? Проверьте свой ответ, выполнив программу на своем компьютере.

2. Измените последний вариант программы калькулятора так, чтобы возврат из функции `float_error` осуществлялся выполнением функции `longjmp`.

3. Приведите пример программы, в которой необходимо или целесообразно игнорировать сигналы, т.е. такой программы, в которой функция обработки сигналов SIG_IGN была бы ассоциирована с одним или несколькими сигналами.

Глава 8

Параллельное программирование

Параллельной называется программа, построенная из последовательных компонентов, называемых процессами, которые выполняются параллельно. Параллельное программирование является желательным по многим причинам [38, 28]:

средства параллельного программирования обеспечивают удобные обозначения и концептуальную ясность при разработке операционных систем, систем реального времени, систем управления базами данных и программ моделирования, т.е. таких систем и программ, в которых многие события могут происходить одновременно;

алгоритмы, содержащие параллелизм в своей основе, наилучшим образом могут быть выражены на языке, имеющем средства для явного выражения параллелизма; при отсутствии таких средств может быть потеряна структура алгоритма;

параллельное программирование может сократить время выполнения программ, так как для параллельного выполнения различных частей программы можно использовать истинно многопроцессорное оборудование.

Программист, не имеющий в своем распоряжении языка со средствами параллельного программирования, будет избегать таких решений поставленных задач, которые допускают параллелизм.

Язык Си не обеспечивает средств для параллельного программирования, однако писать параллельные программы на языке Си можно:

При программировании на языке Си в системе UNIX параллелизм и связь между процессами, аналогичными вводу-выводу, обеспечивается вызовами ядра системы UNIX, а не обеспечивается средствами самого языка Си. Тем не менее такое сочетание может рассматриваться как прекрасный пример активно используемого языка, обеспечивающего параллелизм на практике [73].

Параллельные программы, написанные на языке Си, используют библиотечные функции, которые для обеспечения параллелизма обращаются к операционной системе. Деннис Ритчи, разработчик языка Си, не собирался включать в язык средства параллельного программирования, объясняя это следующими причинами [76]:

1. Одной из целей, поставленных при разработке языка, является сохранение языка относительно небольшим.

2. Обеспечение возможностей параллельного программирования в самом языке неприемлемо из-за излишнего усложнения и так очень трудной задачи разработки языка.

3. Включение средств параллельного программирования в язык нецелесообразно, поскольку привело бы к необходимости сильных допущений об используемой операционной системе; в то же время на практике не исключено сильное отличие этих допущений от реальных возможностей операционной системы.

8.1. ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ СИ В ОПЕРАЦИОННОЙ СИСТЕМЕ UNIX

Параллельная программа на языке Си состоит из двух или более законченных последовательных программ, выполняющихся параллельно. Выполняющаяся программа называется в языке Си *процессом*. Взаимодействие процессов состоит из передачи данных и синхронизации.

В программу на языке Си параллелизм вводится выполнением следующих действий:

1. Вызовом функции `fork` создается точная копия процесса.

2. С помощью функции `exec` этот новый процесс преобразуется в нужный процесс, выполняемый с перекрытием образа некоторой другой программы (выполняемой версии программы, т.е. файла `a.out`, полученного после компиляции).¹

3. Межпроцессорная связь устанавливается с помощью специального средства системы UNIX - *канала* (`pipe`). Неформально канал может рассматриваться как средство преобразования выходных данных одного процесса во входные данные другого процесса без использования промежуточных файлов. Хотя каналы являются основным средством межпроцессорной связи, могут использоваться также сигналы и файлы.

¹ Перекрытие нового процесса с образом другой программы является, строго говоря, не обязательным, поскольку этот новый процесс может продолжать выполнение исходной программы - возможно, некоторой другой секции программы.

8.2. СОЗДАНИЕ ПРОЦЕССА С ПОМОЩЬЮ БИБЛИОТЕЧНОЙ ФУНКЦИИ `fork`

Новый процесс создается выполнением системного вызова `fork`; созданный таким образом процесс почти, но не совсем идентичен процессу, который выполняет системный вызов `fork`. Этот новый процесс (*порожденный процесс*) и процесс, его создавший (*порождающий процесс*), имеют следующие отличия:

1. Порождающий и порожденный процессы имеют разные номера идентификации процесса (*pid - process identification number*).

2. Значения, возвращаемые функцией `fork`, для обоих процессов различны. Значение, возвращаемое функцией `fork` в порождающий процесс, является номером идентификации порожденного процесса, а значение, возвращаемое в порожденный процесс, равно 0.

3. Порождающий процесс получает все открытые файлы и каналы процесса, его порождающего; однако порожденный процесс имеет собственные копии дескрипторов этих файлов и каналов.

Например, успешное выполнение системного вызова `fork` в операторе *switch*

```
switch (pid = fork()) {
case 0: /* порожденный процесс */
    .
    .
    .
case -1: /* порождающий процесс: неудача в */
        /* создании нового процесса */
    .
    .
    .
default: /* порождающий процесс: новый */
        /* процесс создан */
    .
    .
    .
}
```

создает новый процесс; порождающий процесс будет продолжаться при переходе на метку `default`, а альтернатива `case 0` дает начало порожденному процессу. При невозможности создания нового процесса функция `fork` возвращает значение -1 и процесс, вызвавший функцию `fork`, продолжает выполнение с альтернативы с меткой `case -1`.

С помощью библиотечной функции `wait` можно перевести порождающий процесс в состояние ожидания завершения порожденного процесса.

Существует два метода организации параллельных программ, в которых порождающий и порожденный процессы должны выполнять различные задачи:

1. Порожденный процесс выполняет ту же программу, что и порождающий процесс. Однако процессы выполняют различные части программы. Например, порожденный процесс выполняет только коды для альтернативы с меткой case 0, а порождающий процесс - только коды для альтернативы с меткой default.

2. Порождающий процесс продолжает выполнение исходной программы, а порожденный процесс выполняет другую программу, которая перекрывается с исходной.

Первый метод приводит к нерациональному использованию памяти, так как в ней содержатся коды программы, которые не будут выполняться; более того, использование такого метода может привести к непредсказуемым ошибкам из-за того, что как порождающий, так и порожденный процессы могут изменять данные одного и того же файла и выполнять операции чтения и записи с одним и тем же каналом. Каждый процесс содержит лишние коды и имеет доступ к файлам и каналам, относящимся к другому процессу, что делает параллельную программу немодульной.

Второй метод требует несколько больших усилий для установления связи между процессами; к счастью, большая часть этой работы является рутинной и несложной. Достоинство второго метода состоит в том, что он позволяет независимо написанным программам выполняться параллельно и в то же время взаимодействовать друг с другом в достижении общей цели. Приведенные ниже примеры организации параллельных программ базируются на втором методе.

8.3. БИБЛИОТЕЧНАЯ ФУНКЦИЯ `exec1` ДЛЯ ПЕРЕКРЫВАЮЩИХСЯ ПРОЦЕССОВ

Если порожденный процесс уже создан, то с помощью библиотечной функции `exec1` можно выполнить желаемую программу *P*. В результате выполнения вызова

`exec1(полное имя программы P, P, 0)`

процесс, выполняющий этот вызов, перекрывается указанной программой *P* (точнее, *P* является файлом, представляющим собой скомпилированную и готовую к выполнению программу на языке Си). Фактический параметр *полное имя программы P* указывает местонахождение программы *P* в файловой системе операционной системы UNIX (в соответствии с терминологией

системы UNIX полное имя является именем маршрута файла P).¹

В качестве примера рассмотрим следующий сегмент программы:

```
switch (pid = fork()) {
case 0: /* порожденный процесс */
    .
    .
    .
    execl("/a1/nhg/acb/count", "count", 0);
    .
    .
    .
case -1: /* порождающий процесс: неудача при */
        /* создании нового процесса */
    .
    .
    .
default: /* порождающий процесс: новый процесс */
        /* создан */
    .
    .
    .
}
```

Этот процесс, созданный с помощью функции `fork`, будет выполнять программу, скомпилированную под именем `count`, с полным именем `/a1/nhg/acb/count`.

8.4. КАНАЛЫ - СИНХРОННЫЕ СРЕДСТВА СВЯЗИ

Канал является однолинейным синхронным средством связи, используемым двумя процессами для обмена данными и синхронизации друг друга. Если информация должна пересылаться в обоих направлениях, то необходимы два канала. Один процесс записывает информацию с одного конца канала, а другой процесс считывает информацию с другого конца этого канала.

Каналы аналогичны файлам, хотя и несколько отличаются от них:

Попытка записи в "полный" канал приводит к приостановке процесса записи до тех пор, пока канал снова не окажется в состоянии принимать данные, т.е. пока

¹ Более детальное описание функции `execl` приведено в приложении А.

данные не будут прочитаны с другого конца канала. Аналогично, попытка чтения из "пустого" канала приводит к приостановке процесса до тех пор, пока канал не окажется "непустым", т.е. пока в канал не будут записаны данные процессом с конца канала, предназначенного для записи. Такие приостановки процессов являются главным средством синхронизации в параллельных программах на языке Си.

В то время как к файлам возможен прямой доступ (при условии, что эти файлы находятся на диске), к каналам разрешен только последовательный доступ.

До создания нового процесса каналы должны быть установлены в порождающем процессе; должны быть назначены *дескрипторы каналов* (аналогично дескрипторам файлов). Дескрипторы канала указывают концы канала для чтения и записи. Как уже упоминалось, порожденный процесс "наследует" каналы, файлы и данные от порождающего его процесса; дескрипторы файлов и каналов в порожденном процессе являются копиями дескрипторов файлов и каналов порождающего процесса.

8.4.1. Установка канала

Каналы устанавливаются выполнением системного вызова `pipe`. В качестве иллюстрации использования канала предположим, что первоначально были назначены дескрипторы файлов 0, 1 и 2 и что они были ассоциированы с потоками `stdin`, `stdout` и `stderr` соответственно; других дескрипторов файлов назначено не было.

Предположим теперь, что `r` определен как двумерный целый массив:

```
int p[2];
```

Тогда системный вызов

```
pipe(p);
```

назначает следующие два дескриптора файлов, т.е. 3 и 4, для использования в качестве дескрипторов канала. Дескриптор файла `p[0]` представляет конец канала, предназначенный для чтения, а дескриптор файла `p[1]` - конец канала, предназначенный для записи. При успешном завершении функция `pipe` возвращает 0, при неудачном завершении - значение -1.

Для записи в канал и для чтения из канала можно использовать системные вызовы `write` и `read`. Кроме того, с каждым концом канала с помощью функции `fdopen` можно ассо-

цировать поток, после чего все функции, такие как `putc` и `getc`, которые используются при передаче данных потоком, могут использоваться и с каналами. Например, конец канала `p`, предназначенный для записи, можно преобразовать в поток с указателем `fp` следующим образом:

```
FILE *fdopen(), *fp;  
.  
.  
.  
fp = fdopen(p[1], "w");
```

Теперь знаки можно записывать в канал, пользуясь функциями записи в поток, например:

```
putc(c, fp);
```

8.4.2. Переназначение стандартного ввода-вывода

Как уже упоминалось, установление связи между двумя процессами, выполняющими различные программы, требует больших усилий. Рассмотрим теперь, как устанавливается эта связь. Предположим, что:

порожденный процесс выполняет независимо написанную программу, которая читает данные из стандартного входного потока `stdin` и записывает их в стандартный выходной поток `stdout`, т.е. использует макрос `getchar` и функцию `printf`;

порождающий и порожденный процессы должны взаимодействовать, используя два канала `p` и `q`, т.е. порождающий процесс обеспечивает данными порожденный процесс через канал `p` и ожидает получения результатов через канал `q`.

Как же устанавливается связь между порожденным и порождающим процессами? Порожденный процесс ожидает получения своих входных данных из потока `stdin`, однако при параллельном выполнении ввод этих данных будет обеспечиваться порождающим процессом через канал `p`; аналогично порожденный процесс должен передавать свои выходные данные в канал `q`, а не в поток `stdout`. Как преодолеть эти различия?

Одним из решений этой проблемы является переназначение стандартного ввода и вывода в порожденном процессе. Стандартный файл ввода делается синонимом конца канала `p`, предназначенного для чтения, т.е. дескриптор файла `0` яв-

ается теперь ссылкой на конец канала *p* для чтения; аналогично дескриптор файла 1 ассоциируется с концом канала *q* для записи. Если каналы *p* и *q* уже установлены, т.е. выполнены вызовы

```
pipe(p);  
pipe(q);
```

то в порожденном процессе дескрипторы стандартных входных и выходных файлов ассоциируются с каналами заменой соответствия между дескрипторами файлов и каналов. Изменение соответствия между дескрипторами файлов и каналов в порожденном процессе не влияет на такое же соответствие в порождающем процессе (в начальный момент порожденный и порождающий процессы имели одно и то же соответствие между дескрипторами файлов и каналов).

В порожденном процессе конец канала *p* для чтения (*p*[1]) и конец канала *q* для чтения (*q*[0]) использоваться не будут. Более того, дескриптор 0 будет использоваться для доступа к концу канала *p* для чтения вместо дескриптора канала *p*[1], а дескриптор 1 будет использоваться для доступа к концу канала *q* для чтения вместо дескриптора канала *q*[0]. Следовательно, все дескрипторы каналов *p*[0], *p*[1], *q*[0] и *q*[1] могут быть закрыты, т.е. освобождена отведенная для них память. Закрытие дескриптора приводит к следующему соответствию между файлом и его дескриптором: *p*_{read} - 0, *q*_{write} - 1, *stderr* - 2.

Закрытие дескрипторов позволяет:

1. Открыть и использовать закрытые дескрипторы где-нибудь для других целей (в системе UNIX [3] одновременно могут быть открыты не более 20 дескрипторов файлов или каналов).¹

2. Предотвратить случайное обращение к дескрипторам, которые использоваться не должны (при попытке использовать закрытый дескриптор будет выдано сообщение об ошибке).

3. На конце канала для чтения обнаружить конец канала (EOF), когда окажутся закрытыми все дескрипторы, используемые при записи в канал; конец канала для чтения не обнаружит конец канала EOF до тех пор, пока есть хотя бы один открытый дескриптор, относящийся к концу канала для чтения, даже если этот дескриптор, возможно, никогда не будет использован для записи в канал.

Для отделения дескриптора файла 0 от стандартного файла ввода, ассоциирования его с концом канала *p* для чтения и

¹ Число дескрипторов файлов или каналов, которые могут быть открыты одновременно, зависит от конкретной реализации системы.

последующего закрытия дескриптора `p[0]` используется следующая последовательность системных вызовов [6]:

```
close(0);    /* дескриптор файла 0 закрывается в */
             /* связи с закрытием ассоциированного */
             /* с ним файла, т.е. файла стандарт- */
             /* ного ввода */
             /*
dup(p[0]);   /* наименьший доступный дескриптор */
             /* файла, т.е. 0, и p[0], т.е. конец */
             /* канала "p" для чтения, становятся */
             /* синонимами */
             /*
close(p[0]); /* дескриптор p[0] закрывается, так */
             /* как ни один процесс не будет */
             /* читать из канала "p" прямо, */
             /* используя дескриптор канала p[0]; */
             /* чтение будет выполняться с исполь- */
             /* зованием дескриптора 0 */
             /*
```

Функции для чтения из стандартного файла ввода `stdin`, такие как `scanf` и `getchar`, могут теперь использоваться для чтения из канала `p`.

Точно так же дескриптор 1 ассоциируется с концом канала `p` для чтения с помощью следующей последовательности системных вызовов:

```
close(1); dup(q[1]); close(q[1]);
```

Для обеих вышеприведенных последовательностей системных вызовов необходимо, чтобы дескрипторы 0 и 1 были открыты; если это не так, то попытка выполнить системный вызов `close` приведет к ошибке.

8.5. ПРИМЕРЫ

Параллельное программирование на языке Си будет проиллюстрировано на двух простых примерах. В первом примере запрос на копирование файла приводит к созданию процесса, фактически копирующего файл, в то время как порождающий процесс продолжает выполнение, не ожидая завершения копирования файла порожденным процессом; т.е. копирование файла выполняется параллельно с процессом, пославшим запрос на копирование. Между порождающим и порожденным процессами в данном случае нет взаимодействия. Во втором примере подсчет знаков для порождающего процесса выполняется созданным для этой цели порожденным

процессом. Двусторонняя связь между порождающим и порожденным процессами устанавливается с помощью двух каналов: один канал используется порождающим процессом для пересылки знаков порождающему процессу; другой канал служит для передачи порождающему процессу подсчитанного порожденным процессом числа знаков.

8.5.1. Асинхронное, или параллельное, копирование файла

Напишем функцию, которая выполняет копирование одного файла в другой асинхронно, т.е. вызвавшая функцию асору программа продолжает работу, не дожидаясь окончания копирования файла. Функция асору проверяет возможность доступа к обоим файлам, а затем создает новый процесс, выполняющий копирование параллельно с работой вызвавшей ее программы. Этот новый процесс выполняет программу сору, для которой указывается ее полное имя /a1/nhg/asb/сору.

Абстрактное описание программы асору имеет следующий вид:

*проверить, что копируемый файл и файл, куда помещается копия, открыты для чтения и записи соответственно
создать процесс для фактического выполнения копирования
и вернуться в вызывающую программу, не дожидаясь
окончания этого процесса.*

Текст функции асору приведен ниже:

```
/*-----*/  
/* асору: асинхронное копирование файла */  
/*-----*/
```

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

```
int асору(src, dest) /* исходный файл - "src", */  
/* копия файла - "dest"; */  
/* возвращается значение -1 */  
/* в случае ошибки и значе- */  
/* ние 0 при нормальном */  
/* завершении */
```

```
char *src, *dest;  
{  
    FILE *open(), *fs, *fd;  
    int fclose();  
    int fork(), pid;
```

```

if ((fs = fopen(src, "r")) == NULL) {
    printf("асору: файл %s открыть нельзя\n", dest);
    return -1;
}
if ((fd = fopen(dest, "w")) == NULL) {
    printf("асору: файл %s открыть нельзя\n", dest);
    fclose(fs);
    return -1;
}
fclose(fs);
fclose(fd);

switch (pid = fork()) {
case 0: /* порожденный процесс, который будет */
        /* копировать файл одновременно с */
        /* программой, вызывающей эту функцию */
        execl("/a1/nhg/acb/сору", "сору", src, dest, 0);
        /* при успешном выполнении функции execl */
        /* возврата из нее не будет */
        printf("асору: ошибка в execl\n");
        return -1;
case -1: /* порождающий процесс не в состоянии */
        /* создать новый процесс */
        printf("асору: создать процесс нельзя\n");
        return -1;
default: /* порождающий процесс - успешное */
        /* завершение */
}
}

/*-----*/

```

В вышеприведенной программе с целью небольшой оптимизации в операторе *switch* вместо оператора *break* был использован оператор *return*.

Программа *сору*, которая выполняется порожденным процессом, созданным в программе *асору*, является результатом компиляции файла *сору.с*:

```

/*-----*/
/* сору: копирование файла "а" в файл "b" */
/*-----*/

#include <stdio.h>

main(argc, argv) /* имена файлов передаются как */
                 /* аргументы из командной строки. */

```

```

/* Файл в "argv[1]" копируется в */
/* "argv[2]" */
int argc;
char *argv[];
{
    FILE *fopen(), *fs, *fd;
    int fclose();
    int c;

    fs = fopen(argv[1], "r"); /* открытие файла для */
                             /* чтения */
    fd = fopen(argv[2], "w"); /* открытие файла для */
                             /* записи */

    while ((c = fgetc(fs)) != EOF)
        fputc(c, fd);

    fclose(fs);
    fclose(fd);
}

/*-----*/

```

Для получения функции сору в выполняемом виде файл сору.с компилируется по команде

```
cc -o сору сору.с
```

8.5.2. Подсчет неформатирующих знаков в файле

Предположим, что файл содержит команды форматирования и текст. Все эти команды начинаются с точки в первой позиции строки и заканчиваются знаком перевода строки. Требуется написать программу для подсчета знаков в этом тексте, т.е. число знаков в файле минус число знаков в форматизирующих командах.

В решении этой задачи используются два процесса: процесс `textcount` читает текстовый файл и при передаче текстовых строк второму процессу `count` удаляет из них команды форматирования; процесс `count` подсчитывает знаки и по достижении конца текста посылает общее число знаков первому процессу. Для передачи знаков текста функции `count` используется канал `p`. Канал `q` служит для пересылки функции `textcount` общего числа знаков.

Текст функции `count` имеет следующий вид:

```

/*-----*/
/* count: определяет общее число знаков в файле */
/*-----*/

#include <stdio.h>

main()
{
    int count = 0;

    while (getchar() != EOF)
        count++;
    printf("%d\n", count);
}

/*-----*/

```

Программа count вводит данные из файла stdin и выводит результаты в файл stdout. Следовательно, процесс, выполняющий функцию count, должен переназначить ввод и вывод так, чтобы ввод данных функцией count выполнялся из канала p, а вывод - в канал q.

```

/*-----*/
/* textcount: подсчитывает число знаков в файле */
/*           после удаления из него всех команд */
/*           форматирования текста, строк со */
/*           знаком "." в первой позиции */
/*-----*/

#include <stdio.h>

#define R 0 /* stdin и индекс для конца */
              /* канала для чтения */
#define W 1 /* stdout и индекс для конца */
              /* канала для записи */

#define TRUE 1
#define FALSE 0
#define PERIOD '.'

main()
{
    int pid; /* номер идентификации процесса */
    int p[2], q[2]; /* возвращается функцией fork; */
                    /* канал "p" предназначен для */
                    /* пересылки текста с удаленными */
                    /* командами форматирования */

```

```

        /* функции "count"; канал "q"      */
        /* будет использоваться для полу-  */
        /* чения от функции "count" общего */
        /* числа знаков */
FILE *fdopen(), *fp;

int c;
int newline = TRUE; /* начальное значение в начале */
                    /* новой строки */
int total;

/* создать новый процесс для подсчета знаков */
pipe(p); /* p[R] - конец для чтения, p[W] - */
        /* конец для записи */
pipe(q); /* q[R] - конец для чтения, q[W] - */
        /* конец для записи */
switch (pid = fork()) {
case 0:
    /* установить каналы */
    /* порожденный процесс будет читать из кана- */
    /* ла "p" и записывает в канал "q"; следова- */
    /* тельно, конец канала "p" для записи и */
    /* конец канала "q" для чтения закрыты; */
    /* стандартный ввод порожденного процесса */
    /* и конец канала "p" для чтения являются */
    /* синонимами; то же самое сделано для конца */
    /* канала "q" для чтения и стандартного */
    /* вывода */

    close(p[W]);
    close(R); dup(p[R]); close(p[R]);
        /* стандартный ввод порожденного */
        /* процесса и p[R] - синонимы */
    close(q[R]);
    close(W); dup(q[W]); close(q[W]);
        /* стандартный вывод порожденного */
        /* процесса и q[W] - синонимы */

    execl("/a1/nhg/acb/count", "count", 0);
    printf("textcount: ошибка при вызове");
    exit(1);
case -1: /* порождающий процесс не может */
        /* создать новый процесс */
    printf("textcount: неудача при вызове fork");
    exit(1);
default:
    close(p[R]); close(q[W]);
    fp = fdopen(p[W], "w");

```

```

        /* преобразование в поток */
    }

    /* удалить команды форматирования из текста и пос- */
    /* лать его функции "count"; команды форматирования */
    /* начинаются с точки (знак PERIOD) в позиции 1 */
    while ((c = getchar()) != EOF) {
        switch (newline) {
            case TRUE:
                if (c == "\n") /* пустая строка */
                    putc(c, fp);
                else if (c == PERIOD)
                    /* пропустить строку */
                    while ((c=getchar())!=EOF&&c!='\n')
                        ;
                else {
                    putc(c, fp);
                    newline = FALSE;
                }
                break;
            default:
                putc(c, fp);
                if (c == '\n')
                    newline = TRUE;
        }
    }

    fclose(fp); /* чтобы процесс мог воспринять */
               /* EOF на конце канала для чтения, */
               /* должен быть закрыт конец канала */
               /* для записи */

    /* после того, как введены все данные из файла */
    /* стандартного ввода, ввод из канала "q" может */
    /* быть переназначен на файл стандартного ввода, */
    /* т.е. стандартный ввод и q[R] делаются синони- */
    /* мами */
    close(R); dup(q[R]); close(q[R]);

    scanf("%d", &total);
    printf("Общее число знаков %d\n", total);

    exit(0);
}

/*-----*/

```

8.6. ЗАДАЧИ

1. Измените пример с параллельным подсчетом знаков, приведенный в последнем подразделе, так, чтобы вывод функции `count` направлялся прямо в файл вместо обратной пересылки функции `textcount` через канал.

2. (Для читателей, знакомых с системой UNIX): программу для параллельного подсчитывания знаков можно написать проще и в более общем виде, если использовать для канала обозначение `|`, имеющееся в командном языке системы UNIX. Например, если программа `strip` удаляет команды форматирования, а программа `count` подсчитывает число введенных ею знаков, то для параллельной работы этих двух программ достаточно дать команду

```
strip <файл-с-данными | count
```

Сравните этот вариант с вышеприведенным в тексте книги. Почему решение без использования обозначения командного языка для канала намного сложнее?

3. Функции `tbl`, `eqn` и `troff` получают данные из файла `stdin` и выводят результаты в файл `stdout`. Напишите программу, которая выполняет эти три программы параллельно, так, чтобы вывод функции `tbl` был вводом для функции `eqn`, а вывод функции `eqn` был вводом для функции `troff`.

Глава 9 Препроцессор языка Си

Препроцессор используется для обработки текста программы до этапа ее компиляции. Обычно препроцессоры служили средством расширения языков с целью обеспечения дополнительных возможностей. Несмотря на бесчисленное множество препроцессоров, созданных для расширения возможностей языков программирования, все они были нестандартными; для некоторых языков, например для языков общего назначения ПЛ/1 и Си, препроцессоры поставлялись как часть их стандартной среды. Препроцессор для языка Си обеспечивает средства для определения макросов (определение констант является особым случаем), включение файлов и условную компиляцию. Препроцессор языка Си вызывается автоматически при обращении к компилятору по команде `cc`.

Программа может быть обработана только препроцессором без компиляции, если в команде `cc` указать ключ `-E`:

Результат работы препроцессора помещается в поток стандартного вывода stdout. Обработка программы препроцессором без компиляции позволяет программисту проанализировать действие определений препроцессора и макровывозов.

Инструкции препроцессора обычно отличаются от инструкций соответствующего языка. Например, инструкции препроцессора начинаются со знака # в первой позиции строки. Инструкции препроцессора языка Си могут появляться в любом месте программы, а их действие распространяется до конца файла, если оно не прекращено другими инструкциями.

9.1. МАКРООПРЕДЕЛЕНИЯ И МАКРОВЫЗОВЫ

Макросредства позволяют поставить в соответствие идентификатору текстовую строку; все последующие появления в тексте этого идентификатора заменяются соответствующей строкой. Макроопределения препроцессора языка Си имеют две формы, простую и параметризованную:

```
#define идентификатор строка-замены
#define идентификатор( $x_1, x_2, \dots, x_n$ ) строка-замены
```

где *строка-замены* может содержать идентификаторы, ключевые слова, разделители, такие как круглая или прямоугольная скобка, или строки знаков, не содержащие каких-либо разделителей. *Строка-замены* может быть продолжена на следующую строку текста добавлением знака обратной дробной черты \ в конце продолжаемой строки.

9.1.1. Макровывозы

Каждое появление имени макроса в тексте означает макровывоз. Строка замены сканируется еще раз с целью обнаружения в ней макровывозов. Заметим, что завершение рекурсивных макросов обеспечить невозможно, так как невозможно в макроопределение включить условную инструкцию препроцессора; включение имени макроса в качестве подстроки большей строки не является макровывозом.

9.1.2. Простые макроопределения

Макроопределение в первой форме обеспечивает замену каждого *идентификатора строкой замены* для всех идентификаторов, расположенных после макроопределения. Например, после того, как макроопределения

```
#define NULL 0
#define EOF (-1)
#define GET  getc(stdin)
```

будут обработаны препроцессором, каждый из идентификаторов NULL, EOF и GET будет заменен соответствующей правой частью макроопределения, в частности, во всех местах, где имеется идентификатор GET, он будет заменен строкой

```
getc(stdin)
```

Макросы могут применяться и для ограниченных изменений в синтаксисе языка. Например, макроопределения

```
#define begin {
#define end   }
```

могут придать программе на языке Си схожесть с программой на Паскале. С использованием этих определений оператор while может быть записан как

```
while (e)
begin
.
.
.
end
```

9.1.3. Параметризованные макроопределения

Вторая форма макроопределения является параметризованным вариантом первой формы. Перед заменой *идентификатора строкой замены* при каждом появлении формального параметра в строке замены вместо него подставляется соответствующий формальный параметр. Фактические параметры в макросе должны быть разделены запятыми. Несколько примеров параметризованных макроопределений приведены ниже:

```
#define getchar()  getc(stdin)
#define putchar(x)  putc(x,stdout)
#define MAX(x,y)    ((x)>(y)?(x):(y))
#define MIN(x,y)    ((x)<(y)?(y):(x))
#define UPPER(c)    ((c)-'a'+'A')
/* знак с должен быть в */
/* нижнем регистре      */
#define LOWER(c)    ((c)-'A'+'a')
```

```
/* знак с должен быть */  
/* в верхнем регистре */
```

Определения макросов `getchar` и `putchar` взяты из файла `stdio.h`; макросы `MAX` и `MIN` возвращают максимальное и минимальное значения параметров соответственно; макросы `UPPER` и `LOWER` возвращают знаки нижнего или верхнего регистра соответственно. (Причины включения такого большого числа скобок в определения `MIN`, `MAX`, `UPPER` и `LOWER` объясняются в следующем подразделе.)

Вызов параметризованных макросов аналогичен вызову функции. В качестве примера рассмотрим фрагмент программы с параметризованным макросом:

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

Препроцессор преобразует этот фрагмент программы в следующий текст:

```
while ((c = getc(stdin)) != EOF)  
    putc((c)-'a'+'A'), stdout);
```

9.1.4. Определение констант

Наиболее часто препроцессор языка Си используется для определения констант; для этой цели достаточно простых макроопределений. Определения констант обычно задаются в виде

```
#define имя-константы литерал или имя-константы  
#define имя-константы (выражение с постоянными значениями)
```

Например:

```
#define NULL      0  
#define EOF      (-1)  
  
#define TRUE      1  
#define FALSE     0
```

Если значение определяемой константы задается выражением с постоянным значением, то имеет смысл заключить его в скобки. Например, рассмотрим определение константы

#define E 5+10

Использование константы E в выражениях может привести к неожиданным результатам. Например, хотя значение выражения

E+10

равное 25, будет вычислено правильно, значение выражения

E*10

не будет равно 150, как предполагалось; это значение окажется равным 105, так как в этом выражении E будет заменено строкой 5+10, в результате чего получится выражение

5+10*10

Эта проблема возникает из-за непонимания препроцессором языка Си определений констант, фактически препроцессор о языке Си многого совсем не знает. Определения констант являются только макроопределениями.

9.1.5. Оперативная генерация кода

Выполнение вызова функции требует дополнительных затрат времени и памяти, так как при вызове функции необходимо сохранить содержимое регистров, скопировать значения аргументов функции, выполнить переход к телу функции и вернуться обратно в вызывающую программу. Для маленьких функций, особенно если к ним имеется много обращений, такие затраты могут оказаться существенными. Следовательно, в таких случаях у программиста может возникнуть желание отказаться от использования функций несмотря на их преимущества (такие как абстрагирование управления) и вручную заменить каждый вызов функции ее телом. Такая ручная замена имеет очевидные недостатки; например, текст программы становится трудно читать, понимать и модифицировать. Однако мысль о ручной замене может и не возникнуть, если можно сообщить компилятору, что вызовы некоторых функций должны быть заменены соответствующими телами функций, т.е. обеспечена возможность *оперативной генерации кода (in-line code generation)*. Такая возможность в языке Си реализуется с помощью макросов.

Выше были даны определения

```
#define UPPER(c)    ((c)-'a'+'A')
#define LOWER(c)    ((c)-'A'+'a')
```

которыми идентификаторы UPPER и LOWER определяются как макросы, а не как функции, что оказывается удобным, поскольку в данном случае решаются очень простые задачи, а код, полученный в результате оперативной генерации, требует меньше памяти, чем вызов функций.

Важно также отметить, что в некоторых случаях оперативная генерация кода может привести в существенному увеличению памяти, требуемой для размещения самой программы.

9.1.6. Удаление макроопределений

Инструкцией препроцессора

```
#undef идентификатор
```

можно отменить определение *идентификатора*. Удаление определения идентификатора можно использовать для управления *условной компиляцией* (рассматриваемой в разд. 9.3) и для предотвращения сообщений препроцессора о повторных определениях.

9.2. ВКЛЮЧЕНИЕ ФАЙЛОВ В ПРОГРАММУ

С помощью инструкции include в программу на языке Си можно включить текст произвольного файла. Такая возможность позволяет общие описания констант, данных, типов и функций хранить в отдельных файлах. Эти общие описания и определения могут затем использоваться во многих программах одним или несколькими программистами. Помещение общих описаний и определений в различные файлы с последующим включением их в программы является общеупотребительным элементом стиля при программировании на языке Си. В качестве примера можно привести включение описаний стандартного файла ввода-вывода stdio.h.

Инструкция для включения в программу текста файла include имеет две формы. Первая форма вида

```
#include "fname"
```

предписывает препроцессору до этапа компиляции программы заменить инструкцию include содержимым файла fname из каталога, содержащего файлы с исходным текстом; если препроцессор не находит файла в этом каталоге, то он ищет файл в

стандартных или заранее указанных местах. Инструкция, заданная во второй форме

```
#include <fname>
```

выполняется аналогично, за исключением того, что препроцессор не ищет файл `fname` в каталоге с исходными файлами, а рассчитывает на его размещение в стандартных или заранее определенных местах. Например, в системе UNIX стандартное местонахождение многих файлов - каталог `/usr/include`; полное имя файла `fname` имеет вид `/usr/include/fname`.

Инструкции для включения файлов не могут быть вложенными. И, наконец, спецификация стандартных и заранее определенных мест зависит от реализации языка, а не является его частью.

9.3. УСЛОВНАЯ КОМПИЛЯЦИЯ

Условной компиляцией называется выборочная компиляция только тех частей программы, которые удовлетворяют определенным условиям. Например, можно скомпилировать только те части программы, которые необходимы для конкретной версии системы.

Условная компиляция обладает следующими достоинствами:

1. Обеспечивается средство параметризации во время компиляции. Например, такая возможность может быть использована для генерации программ с различной структурой.

2. Обеспечивается эффективное использование оперативной памяти, поскольку во время выполнения нет необходимости держать в памяти неиспользуемые коды программы.

3. Обеспечивается возможность принятия решений во время компиляции, а не во время выполнения. Часто это оказывается более эффективным (хотя и менее гибким) подходом.

Для условной компиляции используется инструкция препроцессора *if*. Она имеет две формы - с частью *else* и без нее:

```
if-заголовок  
текстовые строки для случая "истина"  
#endif
```

и

```
if-заголовок  
текстовые строки для случая "истина"  
#else
```

```
текстовые строки для случая "ложь"  
#endif
```

где *if-заголовок* является управляющей строкой препроцессора, а текстовые строки могут содержать произвольный текст. Управляющая строка препроцессора *if-заголовок* содержит условие, на основе анализа которого производится выбор соответствующих текстовых строк. Управляющая строка препроцессора имеет следующие три формы:

```
#if выражение с постоянными значениями  
#ifdef идентификатор  
#ifndef идентификатор
```

В первой форме управляющей строки условие определяется значением выражения (нуль или не нуль), соответствующим значениям "истина" или "ложь". Во второй форме значение "истина" соответствует условию, когда идентификатор был ранее определен (и в дальнейшем это определение не отменено) с помощью инструкции *define*; в противоположном случае принимается значение "ложь". В третьей форме значение "истина" соответствует условию, когда идентификатор ранее не был определен (или определен, но затем его определение было отменено) с помощью инструкции *define*; в противоположном случае принимается значение "ложь".

Примером условной компиляции могут служить следующие инструкции:

```
#ifndef MAX_STK_SIZE  
#define MAX_STK_SIZE 128  
#endif
```

Эти инструкции обеспечивают для идентификатора `MAX_STK_SIZE` значение по умолчанию, если оно не задано пользователем.

Другой пример условной компиляции - определение идентификатора `BUFSIZ` (пример взят из файла `stdio.h` [3]):

```
#if u370  
#define BUFSIZ 4096  
#endif  
#if vax || u3b  
#define BUFSIZ 1024  
#endif  
#if pdp11  
#define BUFSIZ 512  
#endif
```

Значение константы BUFSIZ зависит от того, какой из идентификаторов u370, vax, u3b или pdp11 был определен (в предположении, что определен только один такой идентификатор).

В качестве последнего примера рассмотрим описание типа FILE (также взятое из файла stdio.h):

```
typedef struct {
    #if vax || u3b
        int _cnt;
        unsigned char *_ptr;
    #else
        unsigned char *_ptr;
        int _cnt;
    #endif
    unsigned char *_base;
    char _flag;
    char _file;
} FILE;
```

Если определен идентификатор vax или u3b, то тип FILE будет определен как

```
typedef struct {
    int _cnt;
    unsigned char *_ptr;
    unsigned char *_base;
    char _flag;
    char _file;
} FILE;
```

иначе определение типа FILE будет иметь следующий вид:

```
typedef struct {
    unsigned char *_ptr;
    int _cnt;
    unsigned char *_base;
    char _flag;
    char _file;
} FILE;
```

Заметим, что оба варианта структуры типа FILE состоят из одних и тех же компонентов, differing лишь порядком первых двух компонентов. Это описание использует тот факт, что адреса компонентов структуры расположены по возрастанию в соответствии с порядком расположения описаний этих компонентов при чтении их слева направо [77]. По утверждению моего коллеги Джона Линдермана, на большинстве машин можно обеспе-

чить более эффективное обращение к первому элементу структуры, чем к другим элементам. Таким образом, вероятно, что первый вариант структуры типа FILE приводит к более эффективной программе на компьютерах типа VAX(DEC) и 3B(AT&T), в то время как второй вариант более эффективен для машин фирмы IBM.

9.4. ЗАКЛЮЧИТЕЛЬНЫЕ ЗАМЕЧАНИЯ

Хотя препроцессоры и расширяют возможности языков программирования, их использование не лишено недостатков:

использование препроцессоров требует дополнительного просмотра текста программы и, как следствие, добавочного времени;

сообщения об ошибках компилятора относятся не к тексту исходной программы, а к тексту, полученному препроцессором. Для устранения этой проблемы препроцессор языка Си применяет специальные приемы.

9.5. ПРИМЕР ИСПОЛЬЗОВАНИЯ ПРЕПРОЦЕССОРА

Для иллюстрации возможностей препроцессора языка Си определим обобщенную функцию. Обобщенная функция является шаблоном для обычной функции. В дополнение к обычным параметрам обобщенная функция позволяет также задавать в качестве параметров типы данных и имена функций. Задавая соответствующие параметры, из обобщенной функции можно получать обычные функции. В языке Си обобщенные функции могут быть реализованы с помощью препроцессора.

Использование обобщенных средств в языках программирования имеет следующие преимущества [28]:

1. *Уменьшение трудоемкости программирования*: легче написать и сопровождать одну обобщенную функцию, чем несколько обычных функций.

2. *Упрощение работы с программой*: программы становятся меньше, так как для написания нескольких обычных функций необходима только одна обобщенная функция.

Напишем обобщенную функцию (фактически макрос) для перестановки элементов. Эта обобщенная функция будет затем использована для создания обычных функций. Например, предположим, что обобщенной функции перестановки присвоено имя `GENERIC_SWAP`. Тогда обычные функции `swap_int` и `swap_float` для перестановки элементов типа `int` и `float` можно создать с помощью макровыводов

```
GENERIC_SWAP(swap_int, int)
GENERIC_SWAP(swap_float, float)
```

Функция `GENERIC_SWAP` определяется как

```
#define GENERIC_SWAP(NAME, ELEM_TYPE) void NAME(a, b)\
    ELEM_TYPE *a, *b;\
    {\
        ELEM_TYPE t;\
        t = *a;\
        *a = *b;\
        *b = t;\
    }
```

С помощью приведенных выше макровывозов могут быть созданы следующие обычные функции:

```
void swap_int(a, b)
int *a, *b;
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
```

```
void swap_float(a, b)
float *a, *b;
{
    float t;
    t = *a;
    *a = *b;
    *b = t;
}
```

к которым теперь допускается прямое обращение.

9.6. ЗАДАЧИ

1. Проанализируйте все "за" и "против" определения констант с помощью препроцессора, как в языке Си, и таких средств определения констант, как в языках программирования Ада, Паскаль или Фортран-77.

2. Зачем так много скобок потребовалось в определениях макросов `MAX` и `MIN`? Почему неверны следующие определения?

```
#define MAX(x, y)    x>y?x:y
#define MIN(x, y)    x>y?y:x
```

3. Напишите макрос ADD для генерации сегмента программы, с помощью которого можно сложить элементы двух массивов и присвоить полученные результаты элементам третьего массива. Например, с помощью такого макроса, как

```
ADD(x, y, z, n)
```

можно было бы получить инструкции вида

```
x[i] = y[i] + z[i];
```

для значений переменной *i* в пределах от 0 до *n*-1. *Подсказка:* определите локальную переменную цикла.

4. Когда макросы (для оперативной генерации кода) вместо функций использовать нежелательно?

5. Напишите макроопределения для реализации операторов *if* в форме (как в Алголе):

```
if выражение
then ...
else ...;
```

Можно дать определение в виде

```
#define if    if(
```

или необходимо использовать другое ключевое слово вместо *if*, например:

```
#define IF    if(
```

6. Напишите обобщенную функцию сортировки **GENERIC_SORT**, которой в качестве параметров можно задавать имя функции сортировки, тип элемента массива и операции сравнения. Например, получить обычную функцию `int_sort` для сортировки массива типа `int` с расположением его элементов в невозрастающем порядке можно с помощью макровывоза

```
GENERIC_SORT(int_sort, int, <)
```

Каков результат выполнения макровывоза

```
GENERIC_SORT(int_sort, int, >)
```

в этом случае?

7. Напишите программу, реализующую некоторые из средств препроцессора языка Си, например операторы `define`.

Глава 10

Последний пример

10.1. ПОЛУЧЕНИЕ ИНФОРМАЦИИ ИЗ БАЗЫ ДАННЫХ

Предположим, что имеется база данных, содержащая информацию о сотрудниках небольшой компании (около 50 сотрудников). Напишем функции для работы с этой базой данных.¹ Информация о сотрудниках содержится в файле, который используется функцией `initialize_db` для инициализации базы данных. Этот файл состоит из строк, каждая из которых содержит восемь полей данных, обозначенных следующим образом:

| | |
|------------------------------|---|
| <code>name(19)</code> | фамилия, инициалы; |
| <code>room(7)</code> | номер комнаты; |
| <code>extension#(4)</code> | номер телефона; |
| <code>designation(11)</code> | профессия и должность; |
| <code>companyid(5)</code> | идентификационный номер в компании; |
| <code>signature(3)</code> | личная подпись (трехзначный пароль); |
| <code>logid(3)</code> | идентифицирующая информация в компьютерной системе; |
| <code>mailid(49)</code> | имя или каталог для электронной почты. |

Эти поля в строках данных отделяются друг от друга пробелами; числа в скобках указывают длины полей (в файле эти числа не хранятся).

Список необходимых функций приведен ниже:

| | |
|-------------------------------------|--|
| <code>initialize_db(db_file)</code> | инициализирует базу данных с файлом <code>db_file</code> ; |
| <code>print_db()</code> | распечатывает содержимое базы данных для отладки; |
| <code>emp_ext(n)</code> | возвращает указатель на строку с содержимым <code>n</code> в поле <code>extension</code> ; |
| <code>emp_room(n)</code> | возвращает указатель на строку с содер- |

¹ Поставленная задача опирается на конкретную разработку, а именно, базу данных, представляющую часть системы-прототипа для обработки документов [27]. Эта система была разработана для экспериментального исследования среды учреждения, в котором сотрудники пользовались компьютерным представлением документов, пересылаемых между сотрудниками с помощью электронной почты. Источником информации для заполнения документов и проверки правильности другой информации служила база данных этой системы.

| | |
|------------------------------|--|
| <code>emp_desig(n)</code> | жимым <code>p</code> в поле <code>room</code> ; возвращает указатель на строку с содержанием <code>p</code> в поле <code>designation</code> ; |
| <code>emp_compid(n)</code> | возвращает указатель на строку с содержанием <code>p</code> в поле <code>companionid</code> ; |
| <code>emp_sig(n)</code> | возвращает указатель на строку с содержанием <code>p</code> в поле <code>signature</code> ; |
| <code>emp_logid(n)</code> | возвращает указатель на строку с содержанием <code>p</code> в поле <code>logid</code> ; |
| <code>get_desig(l_id)</code> | возвращает указатель на строку с содержанием поля <code>designation</code> , соответствующим значению <code>l_id</code> ; |
| <code>get_sig(l_id)</code> | возвращает указатель на строку с содержанием поля <code>signature</code> , соответствующим значению <code>l_id</code> ; |
| <code>get_maild(l_id)</code> | возвращает указатель на строку с содержанием поля <code>mailid</code> , соответствующим значению <code>l_id</code> . |

Параметрами этих функций являются знаковые строки. При отсутствии указанного аргумента в базе данных все функции возвращают значение `NULL`. На основе предположений об использовании базы данных можно ожидать, что об одном и том же сотруднике будет сделано несколько запросов подряд. Другими словами, последовательность запросов к базе данных может быть разбита на подпоследовательности, каждая из которых состоит из запросов об одном и том же сотруднике. На основе сказанного можно попытаться оптимизировать процесс поиска информации в базе данных.

10.1.1. Стратегия разработки базы данных

Вследствие малых размеров базы данных она может целиком размещаться в оперативной памяти; данные из файла, содержащего информацию о сотрудниках, будут присваиваться объектам типа `emp`. На эти объекты будут указывать элементы массива `db`. Для представления текущего объема базы данных будет использоваться переменная `size`. Обе переменные `db` и `size` будут глобальными для всех функций, однако с целью ограничения их видимости они будут определены как статические:

```
static emp *db[MAX_DB];
static int size;
```

Для объектов типа `emp` память будет выделяться по мере необходимости. Для поиска в базе данных большинство функций вызывает функцию `search_db`, использующую в качестве ключа имя сотрудника. Эта функция присваивает переменной `sig` (если

это возможно) индекс соответствующего элемента массива db. Следовательно, в большинстве случаев элемент db[cur] указывает на информацию о сотруднике, упомянутом в последнем успешно выполненном запросе. Учитывая группирование запросов, функция search_db сначала проверяет, является ли информация, на которую указывает элемент db[cur], информацией именно о том сотруднике, который указан в запросе. Если ответ положительный, то функция search_db возвращает значение, равное 1; если ответ отрицателен, то функция search_db организует последовательный поиск в базе данных; если информация о рассматриваемом сотруднике содержится в базе данных, то функция search_db устанавливает значение cur равным индексу элемента массива db, указывающего на искомую информацию о сотруднике, и возвращает значение 1, в противном случае функция search_db возвращает значение 0. Переменная cur определяется как

```
static int cur = 0;
```

10.1.2. Функции базы данных

Приведенные ниже функции просты и не требуют дополнительных пояснений:

```
#include <stdio.h>

/* формат записи файла базы данных */

#define LN 20 /* длина поля name + 1 */
/* для нулевого знака */
#define LR 8 /* длина поля room + 1 */
/* для нулевого знака */
#define LE 5 /* длина поля extension + 1 */
/* для нулевого знака */
#define LD 12 /* длина поля designation + 1 */
/* для нулевого знака */
#define LC 6 /* длина поля companyid + 1 */
/* для нулевого знака */
#define LS 4 /* длина поля signature + 1 */
/* для нулевого знака */
#define LL 4 /* длина поля logid + 1 */
/* для нулевого знака */
#define LM 50 /* длина поля mailid + 1 */
/* для нулевого знака */

#define MAX_DB 100 /* максимальный объем */
```

```

/* базы данных (число */
/* записей) */

/* некоторые функции для обработки строк, */
/* автоматически загружаемые из библиотеки "libc" */
char *strcpy();
int strlen(), strcmp();

/* структура emp определяет структуру записи файла */
/* базы данных */

typedef struct {
    char name[LN];
    char room[LR];
    char ext[LE];
    char desig[LD];
    char compid[LC];
    char sig[LS];
    char logid[LL];
    char maild[LM];
} emp;
static emp *db[MAX_DB];
/* db - массив указателей на записи базы */
/* данных */

static int cur = 0;
/* содержит указатель на данные о пос- */
/* леднем запрашиваемом сотруднике; этот */
/* указатель введен для повышения эффек- */
/* тивности программы, так как предпола- */
/* гается, что после ответа на запрос о */
/* сотруднике будет сделано еще несколь- */
/* ко запросов о том же сотруднике. */
/* В начальный момент переменная cur */
/* указывает на данные о первом по спис- */
/* ку сотруднике */

static int size = 0;
/* size содержит число записей в базе данных */

/*-----*/
/* initialize_db(db_file): инициализация базы данных */
/* с помощью файла "db_file" */
/*-----*/

initialize_db(db_file)
char *db_file;

```

```

{
FILE *fopen(), *fp;
char *malloc();    /* выделение памяти */
int i;              /* счетчик записей базы данных */

if ((fp = fopen(db_file, "r")) == NULL) {
    printf("initialize_db: файл %s", db_file);
    printf(" файл открыть нельзя");
    exit(1);
}

for (i = 0; ; i++) {
    if (i == MAX_DB) {
        printf("initialize_db: предупреждение:");
        printf(" максимальное число записей в базе\n");
        printf(" данных = MAX_DB\n");
        break;
    }
    /* выделить память */
    if ((db[i]=(emp *)malloc(sizeof(emp)))==NULL) {
        printf("initialize_db: нет свободной памяти\n");
        exit(1);
    }

    if (fscanf(fp, "%s%s%s%s%s%s%s", db[i]->name,
        db[i]->room,db[i]->ext,db[i]->desig,
        db[i]->compid,db[i]->sig,db[i]->logid,
        db[i]->maild) == EOF)
        break;
}
size = i;
fclose(fp);

}
/*-----*/

/*-----*/
/* print_db: отладочная печать содержимого базы */
/*          данных */
/*-----*/

print_db()
{
    int i;

    for(i = 0; i < size; i++)
        printf("--%s %s %s %s %s %s %s\n",
            db[i]->name,db[i]->room,db[i]->ext,

```



```

        db[i]->desig,db[i]->compid,db[i]->sig,
        db[i]->logid);
    }
/*-----*/

/*-----*/
/* search_db(n): внутренняя функция, устанавливающая */
/* значение переменной "cur" равным */
/* значению индекса элемента массива */
/* "db", указывающего на запись, со- */
/* держащую строку "n" в поле name. */
/* При успешном поиске возвращается */
/* значение 1, при неудачном поиске - */
/* значение 0. */
/*-----*/

static int search_db(n)
char n[];
{
    if (strcmp(n, db[cur]->name) == 0)
        return 1;

    for(cur = 0; cur < size; cur++)
        if (strcmp(n, db[cur]->name) == 0)
            return 1;
    cur = 0;
    return 0;
}

/*-----*/

/*-----*/
/* strsave(s): выделить память для строки "s" и */
/* вернуть указатель на нее */
/*-----*/

char *strsave(s)
char s[];
{
    char *p, *malloc90;

    if ((p = malloc((unsigned) strlen(s) + 1)) == NULL) {
        printf("strsave: не хватает динамической памяти\n");
        exit(1);
    }
    else
        strcpy(p, s);
    return p;
}

```

```

}

/*-----*/

/*-----*/
/* emp_ext(n): возвращает указатель на поле */
/*      extension сотрудника "n"; если "n" не */
/*      не найден, возвращается значение NULL */
/*-----*/

char *emp_ext(n)
char n[]:
{
    return search_db(n)?strsave(db[cur]->ext):NULL;
}

/*-----*/

/*-----*/
/* get_desig(l_id): возвращает указатель на поле */
/*      designation в записи сотрудника */
/*      с значением поля logid, равным */
/*      "l_id"; при неудачном поиске */
/*      возвращает значение NULL */
/*-----*/

char *get_desig(l_id)
char l_id[];
{
    int i;

    for(i = 0; i < size; i++)
        if (strcmp(db[i]->logid, l_id) == 0)
            return strsave(db[i]->desig);
    return NULL;
}

/*-----*/

```

Поскольку функции `emp_room`, `emp_desig`, `emp_compid`, `emp_sig` и `emp_logid` аналогичны функции `emp_ext`, их тексты не приводятся. По той же причине не даются тексты функций `get_sig` и `get_mailid`, аналогичных функции `get_desig`.

10.2. ЗАДАЧИ

1. Функция `strsave` используется для создания копии ответа на запрос, а ее значение является адресом этой копии, который

возвращается программе, сделавшей запрос. Какие неприятности могут возникнуть при такой передаче адреса копии ответа? В чем трудности такой стратегии? Предложите другую стратегию.

2. Предложите способы повышения эффективности функций базы данных.

3. Обеспечьте возможность изменения данных в базе данных, например, добавления новых записей или модификации существующих. Любые изменения данных в какой-то момент должны быть записаны в файл, где информация хранится постоянно. Следовательно, нужна также функция, записывающая содержимое базы данных в файл для постоянного хранения.

4. Многие из функций базы данных очень похожи друг на друга, например, функции `ext`, `room`, `desig`, `compid`, `sig` и `logid`. Используя средства препроцессора, напишите обобщенную функцию, из которой можно получить перечисленные выше функции.

5. Напишите программу для распознавания строк, получаемых с помощью следующей грамматики, записанной в обозначениях расширенной BNF¹ (т.е. программу, определяющую, является ли входная строка правильным выражением)

```
оператор --> переменная = выражение
переменная --> буква { буква-или-цифра }
выражение --> терм { + терм }
терм --> множитель { * множитель }
множитель --> переменная | ( выражение )
```

где *буква* обозначает одну из букв верхнего регистра, а *буква-или-цифра* обозначает одну букву или цифру верхнего регистра.

Подсказка: поставленная задача проще решается с помощью рекурсии. Напишите функцию для каждого термина левой части каждого правила вывода.

Приложение А

Некоторые библиотечные функции

Многие функции, ставшие уже привычными для тех, кто программирует на языке Си, в действительности являются не

¹ Обозначения расширенной BNF используются для определения синтаксиса языков программирования: `[a]` обозначает необязательное присутствие элемента `a`; `{a}` обозначает число появлений элемента `a`, большее или равное 0; `a|b` обозначает один из элементов: `a` или `b`; `a-->b` является правилом вывода `b` из `a`.

частью самого языка, а библиотечными функциями, принадлежащими к его стандартной среде. В этой главе будут описаны некоторые из общих функций, содержащихся в двух библиотеках системы UNIX: в стандартной библиотеке языка Си `libc` и математической библиотеке `libm`. Функции библиотеки `libc` будут описаны более подробно, а о функциях библиотеки `libm` будут приведены лишь краткие сведения. Полный список этих функций можно найти в книгах [5,4].

A1. СИСТЕМНЫЕ ВЫЗОВЫ ОПЕРАЦИОННОЙ СИСТЕМЫ UNIX И СТАНДАРТНАЯ БИБЛИОТЕКА `libc` ЯЗЫКА СИ

Справочные руководства системы UNIX обычно делятся на несколько частей. Функции, описанные во второй части, называются системными вызовами, так как для выполнения своих действий они обращаются к операционной системе UNIX. Функции, описанные в третьей части, также могут обращаться к системе UNIX, но не прямо, а через вызовы функций второй части руководства. Функции, попавшие в части 2 и 3 справочного руководства, включены в библиотеку `libc` системы UNIX; функции из части 3S руководства составляют стандартную среду ввода-вывода (`stdio`).

В спецификациях функций указаны файлы, в которых находятся их описания. Например, в большинстве систем UNIX описания функций стандартного пакета ввода-вывода `stdio.h` даются в файле с именем `/usr/include/stdio.h`; эти описания могут быть включены в программу на языке Си инструкцией препроцессора

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

Приведенные здесь описания функций взяты из книги [4] и переработаны. Из этих описаний было удалено все, что не относится к рассматриваемым вопросам или несущественно, например перечисление ошибок в функциях. Число (возможно, с буквой), указанное после имени функции, является ссылкой на соответствующую часть руководства [4], в котором эта функция описана более детально.

ALARM(2)

(установка времени подачи предупреждающего сигнала для процесса)

Функция `alarm` предписывает установить для вызывающего процесса число секунд реального времени, через которое вызывающему процессу будет послан сигнал `SIGALRM`. Число секунд задается параметром `sec` (см. описание функции `signal`). Спецификация функции имеет следующий вид:

```
unsigned alarm(sec);  
unsigned sec;
```

Запросы на сигнал не образуют стека; успешное выполнение вызова обеспечивает восстановление состояния часов предупреждающего сигнала вызывающего процесса.

Если значение параметра sec равно 0, ранее сделанный запрос на предупреждающий сигнал игнорируется. Функция alarm возвращает значение времени, ранее остававшееся на соответствующих часах вызывающего процесса.

CLOSE(2)

(закрытие дескриптора файла)

Данный дескриптор файла `fildes`, такой, как возвращаемый после вызова функций `open`, `creat`, `dup` или `pipe`, закрывается при обращении к функции `close`, имеющей спецификацию

```
int close(fildes)  
int fildes;
```

При успешном выполнении функция `close` возвращает значение 0; при неудаче (если `fildes` не является правильным дескриптором открытого файла) возвращается значение -1.

При выполнении функции `exit` все открытые файлы автоматически закрываются, однако из-за ограничения числа одновременно открытых для процесса файлов использование функции `close` необходимо в программах, работающих с многими файлами.

CREAT(2)

(создание нового файла или перезапись существующего)

Функция `creat` имеет следующую спецификацию:

```
int creat(path, mode)  
char *path;  
int mode;
```

Вызов функции `creat` обеспечивает создание нового обычного файла или подготовку для перезаписи существующего, заданного маршрутом с помощью параметра `path`.

При успешном завершении функции возвращается неотрицательное число, а именно, дескриптор файла; при неудаче возвращается значение -1.

DUP(2)

(дублирование дескриптора открытого файла)

Данный дескриптор файла `fildes`, полученный при вызове функций `open`, `pipe` или `creat`, дублируется вызовом функции `dup`, имеющей следующую спецификацию:

```
int dup(fildes)
int fildes;
```

Новый дескриптор файла и дескриптор исходного файла относятся:

- к тому же самому открытому файлу (или каналу);
- к тому же самому указателю файла (т.е. оба дескриптора файла разделяют один и тот же указатель файла);
- к тому же самому режиму доступа (чтения, записи или чтения и записи).

Новый дескриптор файла остается открытым при системных вызовах `exec`. Возвращаемый дескриптор файла является наименьшим доступным дескриптором.

При успешном завершении функция `dup` возвращает неотрицательное число, а именно, дескриптор файла. При неудаче (если `fildes` не является правильным дескриптором открытого файла или уже открыто 20 дескрипторов файла) функция возвращает значение `-1`.

EXEC(2)

(выполнение файла)

Для выполнения любой желаемой программы процесс может быть создан с помощью одного из вариантов функции `exec`: `execl`, `execv`, `execle`, `execve`, `execlp` и `execvp`. Эти функции имеют следующие спецификации:

```
int execl(path, arg0, arg1, ..., argn, 0)
char *path, *arg0, *arg1, ..., *argn;
```

```
int execv(path, argv)
char *path, *argv[];
```

```
int execle(path, arg0, arg1, ..., argn, 0, envp)
char *path, *arg0, *arg1, ..., *argn, *envp[];
```

```
int execve(path, argv, envp)
char *path, *argv[], *envp[];
```

```
int execlp(file, arg0, arg1, ..., argn, 0)
```

```
char *file, *arg0, *arg1, ..., *argn;
```

```
int execvp(file, argv)
```

```
char *file, *argv[];
```

Функции `exec` во всех формах превращают вызывающий процесс в новый процесс. Новый процесс создается из файла, представляющего собой скомпилированную и готовую к выполнению программу; этот файл называется "файлом нового процесса". Поскольку вызывающий процесс перекрывается новым процессом, из функции `exec` может не быть возврата при ее успешном выполнении.

Для выполнения программа на языке Си вызывается следующим образом:

```
main(argc, argv, envp)
```

```
int argc;
```

```
char **argv, **envp;
```

где `argc` указывает число аргументов, а `argv` является массивом знаковых указателей на аргументы; по соглашению, значение `argc` должно быть не менее единицы, а первый элемент массива `argv` должен указывать на строку, содержащую имя файла.¹

Аргумент `path` указывает маршрут, идентифицирующий файл нового процесса. Аргумент `file` является именем файла нового процесса.

Аргументы `arg0`, `arg1`, ..., `argn` являются указателями на знаковые строки с нулем в качестве признака конца строки. Эти строки представляют собой список аргументов, доступный новому процессу. По соглашению, должен присутствовать хотя бы аргумент `arg0`, указывающий на ту же строку, что и `path` (или на последний компонент этой строки).

Массив `argv` является массивом знаковых указателей на строки с нулем в качестве признака конца строки. Эти строки составляют список аргументов, доступный новому процессу. По соглашению, в массиве `argv` должен быть хотя бы один элемент, указывающий на ту же строку, что и `path` (или на последний компонент этой строки); массив `argv` заканчивается нулевым указателем.

Аргумент `envp` является массивом знаковых указателей на строки с нулем в качестве признака конца. Эти строки со-

¹ При определении функции `main` могут быть опущены ее аргументы в следующих сочетаниях: `envp` или `argv` вместе с `envp` или все три аргумента одновременно. Тогда при вызове функции `main` значения, передаваемые для опущенных аргументов, не будут доступны в функции `main`.

ставляют среду для нового процесса. Аргумент `envp` завершается нулевым указателем. Для функций `exec1` и `execv` программа времени прогона, иницилирующая работу, помещает указатель на среду вызывающего процесса в глобальную ячейку:

```
extern char **environ;
```

Этот указатель используется для передачи среды вызывающего процесса новому процессу.

Дескрипторы файлов, открытые в вызывающем процессе, остаются открытыми и в новом процессе, за исключением тех дескрипторов, для которых установлены флаги "закрывать по выполнении"; для дескрипторов файлов, которые остаются открытыми, указатели файлов не изменяются.

Сигналы, предписывающие завершение вызывающего процесса, вызовут завершение нового процесса. Сигналы, игнорируемые вызывающим процессом, будут игнорироваться новым процессом.

Если функция `exec` выполняет возврат в вызывающую программу, то это свидетельствует об ошибке; возвращаемое значение в этом случае равно -1.

EXIT, _EXIT(2) (завершение процесса)

Вызов функции `exit` является обычным способом завершения процесса. Функция `exit` имеет следующую спецификацию:

```
void exit(status)
int status;
```

Завершение вызывающего процесса с помощью функции `exit` осуществляется в следующей последовательности: закрываются все дескрипторы файлов, открытые в вызывающем процессе; если порождающий процесс для вызывающего процесса находится в состоянии ожидания, то он извещается о завершении вызывающего процесса. Функция языка Си `exit` до завершения процесса может выполнить действия по очистке. Функция `_exit` со спецификацией

```
void _exit(status)
int status;
```

выполняет все действия по очистке.

FORK(2)

(создание нового процесса)

Функция `fork` используется для создания нового процесса. Ее спецификация имеет следующий вид:

```
int fork()
```

Новый (порожденный) процесс является "точной" копией вызывающего (порождающего) процесса. Существуют, однако, некоторые различия между ними, например:

- порожденный процесс имеет уникальный идентификатор;
- порожденный процесс имеет индивидуальный идентификатор порождающего процесса;
- порожденный процесс имеет собственную копию дескрипторов файлов порождающего процесса. Каждый из дескрипторов файлов порожденного процесса разделяет указатель файла, общий с соответствующим дескриптором файла порождающего процесса.

Если превышено ограничение на общее число процессов в системе или на общее число процессов, разрешенное для одного пользователя, то функция `fork` не в состоянии создать новый процесс и возвращает в порождающий процесс значение -1, свидетельствующее о неудаче. При успешном завершении функция `fork` возвращает значение 0 порожденному процессу, а идентификатор порожденного процесса возвращает в порождающий процесс.

KILL(2)

(посылка сигнала процессу)

Функция `kill` имеет следующую спецификацию:

```
int kill(pid, sig)
int pid, sig;
```

Функция `kill` посылает сигнал `sig` процессу, указанному номером процесса `pid`. При успешном завершении функции возвращается значение 0, при неудачном - значение `r1`.

LSEEK(2)

(установка указателя файла при записи или чтении)

Функция `lseek` используется для установки указателя файла на конкретное место в файле до начала чтения или записи. Спецификация функции имеет следующий вид:

```

long lseek(fildes, offset, whence)
int fildes;
long offset;
int whence;

```

Параметр `fildes` является дескриптором файла, возвращаемым при выполнении системных вызовов `creat`, `open` или `dup`. Указатель файла, ассоциированный с дескриптором файла `fildes`, функция `lseek` устанавливает следующим образом:

| Значение <code>whence</code> | Значение указателя файла |
|------------------------------|--|
| 0 | <code>offset</code> байтов |
| 1 | Текущее значение плюс смещение <code>offset</code> |
| 2 | Размер файла плюс смещение <code>offset</code> |

При успешном завершении возвращается результирующее положение указателя от начала файла, измеряемое в байтах.

Если `fildes` не является открытым дескриптором файла или он ассоциирован с каналом, или значение `whence` не равно 0, 1 или 2, или при заданных условиях может быть получен отрицательный указатель файла, то функция `lseek` возвращает значение -1, сигнализируя о неудачном завершении, а указатель файла остается без изменения. При успешном завершении функция возвращает значение 0.

Некоторые устройства не позволяют установить указатель, для них значение указателя файла остается неопределенным.

OPEN(2)

(открытие файла для чтения или записи)

Функция `open`, используемая при открытии файлов для чтения или записи, имеет следующую спецификацию:

```

#include <fcntl.h>

int open(path, oflag, [, mode])
char *path;
int oflag, mode;

```

Параметр `path` указывает на маршрут, именующий файл. Функция `open` открывает дескриптор указанного файла и устанавливает флаг состояния файла в соответствии с значением параметра `oflag`, которое выбирается из следующего списка (из первых трех флагов может быть использован только один):

| | |
|-----------------------|-----------------------------|
| <code>O_RDONLY</code> | Открытие только для чтения. |
| <code>O_WRONLY</code> | Открытие только для записи. |

O_RDWR
O_NDELAY

Открытие для чтения и записи. Этот флаг может влиять на последующие операции чтения и записи. При открытии очереди запросов к устройствам ввода-вывода с установленными флагами O_RDONLY или O_WRONLY возможны следующие результаты: если установлен флаг O_NDELAY, то открытие файла только для чтения будет сделано без задержки; при попытке открытия файла только для чтения будет выдано сообщение об ошибке, если ни для одного процесса в данный момент не открыт файл для чтения; если флаг O_NDELAY не установлен, то открытие файла только для чтения будет блокировано до тех пор, пока процесс не откроет этот файл для записи; открытие файла только для записи будет блокировано до тех пор, пока процесс не откроет этот файл для чтения.

При открытии файла, ассоциированного с линией связи, возможны следующие результаты:

если флаг O_NDELAY установлен, то возврат из функции open выполняется без ожидания появления носителя с файлом;

если флаг O_NDELAY не установлен, то открытие файла будет блокировано до появления носителя с файлом.

O_APPEND

Если этот флаг установлен, то указатель файла будет устанавливаться перед каждым обращением к файлу для записи.

O_CREAT

Если файл уже существует, то этот флаг не оказывает какого-либо действия. В противном случае идентификатор владельца файла ставится в соответствие идентификатору пользователя, управляющего этим процессом; групповой идентификатор файла ставится в соответствие групповому идентификатору пользователей, управляющих процессом; младшие 12 бит указателя режима использования файла модифицируются следующим образом:

O_TRUNC

O_EXCL

очищаются все наборы битов маски процесса для указателя режима создания файла;
очищается бит сохранения образа текста после выполнения.

Если файл существует, то его длина сокращается до 0, а режим использования файла и пользователь не изменяются. Если установлены флаги **O_EXCL** и **O_CREAT**, то в случае существования файла попытка открыть его приводит к ошибке.

При успешном завершении возвращается неотрицательное число - дескриптор файла. Указатель файла, используемый для отметки текущей позиции в файле, устанавливается на начало файла. Новый дескриптор файла остается открытым все время выполнения системного вызова `exec`. Ни один процесс не может иметь более 20 дескрипторов файла, открытых одновременно.

При неудачном завершении функция `open` возвращает значение `r1` и устанавливается индикатор ошибки `errno`.

PAUSE(2)

(приостановка выполнения процесса до получения сигнала)

Функция `pause` приостанавливает вызывающий ее процесс до получения сигнала. Этот сигнал должен быть таким сигналом, который не может быть игнорирован вызывающим процессом. Функция `pause` имеет следующую спецификацию:

`pause()`

Если полученный сигнал предписывает завершить вызывающий процесс, возврата из функции `pause` не будет. Если сигнал перехвачен вызывающим процессом и произошла передача управления из функции перехвата сигнала (см. описание функции `signal`), вызывающий процесс возобновляет выполнение с точки остановки, получив от функции `pause` значение -1.

PIPE(2)

(создание межпроцессорного канала)

Функция `pipe` имеет следующую спецификацию:

```
int pipe(fildes)
int fildes[2];
```

Эта функция создает механизм ввода-вывода, называемый каналом, и возвращает два дескриптора файла `filides[0]` и `filides[1]`. Дескриптор `filides[0]` открывается для чтения, а дескриптор `filides[1]` - для записи. До того, как канал будет заблокирован для записи, он может выполнить буферизацию до 5120 байтов записываемых данных. Чтение данных, записываемых с использованием дескриптора файла `filides[1]`, обеспечивается с помощью дескриптора файла `filides[0]` по методу "первым пришел - первым ушел".

Если 19 или более дескрипторов файлов уже открыты, обращение к функции `pipe` может оказаться неудачным, поскольку ни для одного процесса не могут быть открыты одновременно более 20 дескрипторов файлов.

При успешном завершении функция `pipe` возвращает значение 0, при неудачном завершении - значение -1.

READ(2) (чтение из файла)

Спецификация функции `read` имеет следующий вид:

```
int read(filides, buf, nbytes)
int filides;
char *buf;
unsigned nbytes;
```

С помощью функции `read` можно прочитать в буфер с указателем `buf` `nbytes` байтов из файла, ассоциированного с дескриптором `filides`. Дескриптор `filides` должен быть получен из системных вызовов `creat`, `open`, `dup` или `pipe`.

Для тех устройств, для которых это возможно, операция чтения начинается с того места файла, которое задается указателем файла, ассоциированным с дескриптором `filides`. После выполнения возврата из функции `read` указатель файла сдвигается на число фактически прочитанных байтов.

Для остальных устройств операция чтения всегда начинается с тех данных файла, которые в этот момент находятся перед читающей головкой устройства. Значение указателя файла, ассоциированное с таким файлом, не определено.

При успешном окончании работы функцией возвращается число байтов, фактически считанное и помещенное в буфер; это число может быть меньше, чем `nbytes`, если этот файл ассоциирован с линией связи или число байтов, оставшееся в файле меньше, чем `nbytes`; при достижении конца файла возвращается значение 0. Обращение к функции `read` заканчивается неудачей, если `filides` не является правильным дескриптором файла, открытого для чтения, или указатель `buf` не попадает в диапазон адресов выделенной памяти. При успешном завершении функ-

ция возвращает неотрицательное число, выражающее количество фактически считанных байтов, при неудачном завершении возвращается значение -1.

SIGNAL(2)

(выбор способа обработки полученного сигнала)

Функция `signal` позволяет вызывающему ее процессу выбрать один из трех возможных способов обработки полученного конкретного сигнала. Она имеет следующую спецификацию:

```
#include <sys/signal.h>

int (*signal(sig, func))()
int sig;
int (*func)();
```

Аргумент `sig` обозначает сигнал, а аргумент `func` - возможный выбор. Аргументу `sig` может быть присвоено одно из следующих значений, кроме значения `SIGKILL`:

| Номер | Имя | Комментарий |
|-------|---------|--|
| 1 | SIGHUP | "Зависание" программы |
| 2 | SIGINT | Прерывание |
| 3 | SIGQUIT | Выход из программы без сохранения данных |
| 4 | SIGILL | Запрещенная инструкция |
| 5 | SIGTRAP | Захват |
| 6 | SIGIOT | Инструкция обращения к супервизору при вводе-выводе |
| 7 | SIGEMT | Инструкция обращения к супервизору |
| 8 | SIGFPE | Особая ситуация при выполнении операции с плавающей точкой |
| 9 | SIGKILL | Прекращение работы |
| 10 | SIGBUS | Ошибка шины |
| 11 | SIGSEGV | Нарушение сегментации |
| 12 | SIGSYS | Ошибочный аргумент в системном вызове |
| 13 | SIGPIPE | Запись для межпроцессорного обмена при отсутствии процесса, читающего данные |
| 14 | SIGALRM | Предупреждение в заданное время |
| 15 | SIGTERM | Программный сигнал завершения |
| 16 | SIGUSR1 | Определенный пользователем |

| | | |
|----|---------|--|
| 17 | SIGUSR2 | сигнал 1 Определенный пользователем |
| 18 | SIGCLD | сигнал 2 Уничтожение подчиненного процесса |
| 19 | SIGPWR | Отказ источника питания |

Аргументу `func` можно присвоить значение `SIG_DEL`, `SIG_IGN` или адрес функции. Этим значениям соответствуют следующие действия:

| Значение <code>func</code> | Действие |
|----------------------------|---|
| <code>SIG_DEL</code> | Завершение процесса по получении сигнала |
| <code>SIG_IGN</code> | Игнорирование сигнала |
| <i>адрес функции</i> | Выполнение указанной функции по получении сигнала |

При успешном завершении функция `signal` возвращает предыдущее значение параметра `func` для указанного сигнала `sig`, при неудачном завершении возвращается значение `pl`.

WAIT(2)

(ожидание остановки или завершения порожденного процесса)

Выполнение функции `wait` вызывает остановку вызывающего ее процесса до получения сигнала, который должен быть обработан (см. описание функции `signal`) или до завершения процесса, порожденного вызывающим процессом. Если порожденный процесс останавливается или завершается до обращения к функции `wait`, то возврат выполняется немедленно. Функция `wait` имеет следующую спецификацию:

```
int wait(stat_loc)
int *stat_loc;

int wait ((int *)0)
```

Если значение указателя `stat_loc` (рассматриваемого как целое число) не равно нулю, то в младших 16 битах ячейки, ссылка на которую делается с помощью указателя `stat_loc`, запоминается информация, называемая состоянием. Биты состояния могут указывать, завершен порожденный процесс или остановлен.¹ Кроме того, если порожденный процесс завершен, то ин-

¹ Остановка процесса имеет место при его выполнении в режиме трассировки [3].

формация о состоянии процесса указывает причину завершения и передает следующую дополнительную информацию порождающему процессу:

1. Если порожденный процесс остановлен, то старшие 8 бит состояния будут содержать номер сигнала, который явился причиной остановки процесса, а в младшие 8 бит будет занесено число 0177.

2. Если порожденный процесс завершен обращением к функции `exit`, то младшие 8 бит состояния будут равны 0, а старшие 8 бит будут содержать младшие 8 бит значения аргумента, переданного порожденным процессом функции `exit`.

3. Если порожденный процесс завершился по сигналу, то старшие 8 бит состояния будут нулевыми, а младшие 8 бит будут содержать номер сигнала, вызвавшего завершение. Кроме того, если установлен седьмой бит группы младших разрядов, то будет создан образ ядра системы.

Если порождающий процесс завершается без ожидания завершения порожденных им процессов, то значение идентификатора порождающего процесса для каждого порожденного процесса устанавливается равным 1.

Обращение к функции `wait` немедленно дает значение -1, если порожденные процессы оказываются несуществующими или указатель `stat_loc` дает ссылку на неверный адрес.

Если выполнение функции `wait` завершается по получении сигнала, то в вызывающий процесс возвращается значение -1. Если функция `wait` завершается при остановке или завершении порожденного процесса, то идентификатор порожденного процесса возвращается в вызывающий процесс. В остальных случаях возвращается значение -1.

WRITE(2) (запись в файл)

Функция `write` имеет следующую спецификацию:

```
int write(fildes, buf, nbytes)
int fildes;
char *buf;
unsigned nbytes;
```

Эта функция записывает `nbyte` байт, находящихся в буфере с указателем `buf`, в файл, ассоциированный с дескриптором файла `fildes`. Дескриптор `fildes` может быть получен из системных вызовов `creat`, `open`, `dup` или `pipe`.

На тех устройствах, на которых это возможно, фактическая запись данных выполняется с того места в файле, на которое установлен указатель файла. При возврате из функции `write` значение указателя файла увеличивается на число фактически записанных байтов. На остальных устройствах запись всегда начинается с текущего положения записывающей головки. Значение указателя файла, ассоциированного с таким устройством, не определено.

Если хотя бы одно из приведенных ниже условий имеет место, то вызов функции `write` закончится неудачей, а указатель файла сохранит прежнее значение:

- `fildes` не является правильным дескриптором файла, открытого для записи;

- была сделана попытка записи в канал, который не был открыт для чтения ни одним процессом;

- была сделана попытка записи в файл, для которого превышен лимит на размер файла для данного процесса или нарушено общее ограничение на максимальный размер файла;

- указатель `buf` установлен на адрес за пределами выделенного объема памяти.

Если с помощью функции `write` делается попытка записи данных в недостаточную по размеру область, то неуместившиеся данные будут потеряны. Например, предположим, что до установленного лимита в файле имеется 20 свободных байтов. Тогда при записи 512 байтов функция `write` запишет только 20. Последующие попытки записи ненулевого числа байтов приведут к неудаче. Запись в заполненный канал будет блокирована до появления свободного места.

При успешном завершении функция `write` возвращает число фактически записанных байтов, при неудачном завершении - значение `p1`.

ABORT(3)

(генерация сигнала завершения ввода-вывода при ошибке)

Функция `abort` вызывает послышку процессу сигнала завершения ввода-вывода. Обычно при таком завершении выдается дамп ядра. Функция `abort` имеет следующую спецификацию:

```
int abort()
```

Из функции `abort` возможен возврат управления, если сигнал `SIGIOT` перехвачен или проигнорирован; в этом случае возвращаемое значение то же самое, что и у системного вызова `kill`.

ABS(3)

(возвращение абсолютного значения целого числа)

Функция `abs`, имеющая спецификацию

```
int abs(i)
int i;
```

возвращает абсолютное значение целого операнда.

ISALPHA, ISUPPER, ISLOWER, ISDIGIT, ISALNUM, ISSPACE,
ISPUNCT, ISPRINT, ISCNTRL, ISASCII(3)
(макросы для классификации знаков)

Функции `isalpha`, `isupper`, и т.д. распознают коды ASCII, представленные целыми значениями, с помощью таблицы кодов. Их определения находятся в файле `/usr/include/ctype.h` и могут быть включены в программу строкой вида

```
#include <ctype.h>
```

Каждый макрос является предикатом, возвращающим ненулевое значение в качестве значения "истина", и нуль в качестве значения "ложь"; его спецификация имеет вид

```
int istype(c)
int c;
```

где вместо слова `type` можно подставить одно из слов `alpha`, `upper`, `lower`, `digit`, `alnum`, `space`, `print`, `punct`, `cntrl` и `ascii`. Макрос `isascii` определен на всех целых значениях; остальные определены только для тех значений, на которых функция `isascii` принимает значение "истина", и для единственного значения EOF, не являющегося кодом ASCII. Ниже перечислены имена функций с указанием утверждения, истинность или ложность которого определяется с помощью функции:

| Функция | Утверждение |
|-------------------------|--------------------------------------|
| <code>isalpha(c)</code> | с является знаком |
| <code>isupper(c)</code> | с является знаком в верхнем регистре |
| <code>islower(c)</code> | с является знаком в нижнем регистре |
| <code>isdigit(c)</code> | с является цифрой |
| <code>isalnum(c)</code> | с является алфавитно-цифровым знаком |
| <code>isspace(c)</code> | с является пробелом, знаком табу- |

| | |
|-------------------------|---|
| | ляции, возврата каретки, перехода на новую строку или смещения формы |
| <code>ispunct(c)</code> | с является знаком пунктуации |
| <code>isprint(c)</code> | с является печатным знаком |
| <code>isctrl(c)</code> | с является знаком удаления (0177) или обычным управляющим знаком (код меньше 040) |
| <code>isascii(c)</code> | с является знаком ASCII с кодом меньше 0200 |

FCLOSE, FFLUSH(3S)

(закрытие потока или освобождение буфера потока)

Функция `fclose` обеспечивает запись данных из буфера для потока `stream` и закрытие этого потока. Спецификация функции имеет следующий вид:

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

```
int fclose(stream)
FILE *stream;
```

При вызове функции `exit` для всех открытых файлов обращение к функции `fclose` выполняется автоматически.

Функция `fflush` обеспечивает запись данных из буфера указанного потока `stream` в соответствующий файл. Функция `fflush` имеет следующую спецификацию:

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

```
int fflush(stream)
FILE *stream;
```

После выполнения функции `fflush` поток `stream` остается открытым.

FOPEN, FREOPEN, FDOPEN(3S)

(открытие потока)

Функции `fopen`, `freopen` и `fdopen` используются для открытия потоков. Они имеют следующие спецификации:

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

```
FILE *fopen(filename, type)
char *filename, *type;
```

```
FILE *freopen(filename, type, stream)
char *filename, *type;
FILE *stream;
```

```
FILE *fdopen(fildes, type)
int fildes;
char *type;
```

Функция `fopen` открывает файл с именем `filename` и устанавливает связь потока `stream` с этим именем. Функция `fopen` возвращает указатель на структуру `FILE`, ассоциированную с потоком `stream`.

Аргумент `filename` указывает на знаковую строку, содержащую имя открываемого файла. Аргумент `type` является знаковой строкой, принимающей одно из следующих значений:

| Значение <code>type</code> | Предписание для функции |
|----------------------------|--|
| <code>r</code> | Открыть для чтения |
| <code>w</code> | Сократить или создать файл для записи |
| <code>a</code> | Пополнить файл; открыть для записи в конец файла или создать файл для записи |
| <code>r+</code> | Открыть для модификации (для чтения и записи) |
| <code>w+</code> | Сократить или создать файл для модификации |
| <code>a+</code> | Пополнить файл; открыть или создать файл для модификации в конце файла |

При обращении к функции `freopen` вместо открытого потока используется файл с указанным именем. Исходный поток закрывается независимо от того, состоится ли это открытие. Функция `freopen` возвращает указатель на структуру `FILE`, ассоциированную с потоком `stream`.

Функция `freopen` обычно используется для назначения ранее открытых потоков, связанных с файлами `stdin`, `stdout` и `stderr`, на другие файлы.

Функция `fdopen` устанавливает связь между потоком `stream` и дескриптором файла, полученным из вызовов `open`, `dup`, `creat` или `pipe`, открывающих файлы, но не возвращающих указатели на поток `stream`, которые необходимы в качестве входных данных для многих программ раздела 3S библиотеки программ. Значение параметра `type` для потока `stream` должно быть согласовано со способом обработки данных файла.

Если файл открыт для модификации, то ввод и вывод данных выполняются с результирующим потоком `stream`. Однако без использования функций `fseek` или `rewind` вывод не может сразу следовать за вводом, а ввод - за выводом без использования функции `fseek` или `rewind` или операции ввода, которая ищет конец файла.

Если файл открыт для пополнения (т.е. параметр `type` имеет значение `a` или `a+`), то информацию, уже имеющуюся в файле, перезаписать невозможно. Для установки указателя файла в любое требуемое положение служит функция `fseek`, но при записи выходных данных в файл текущее значение указателя файла игнорируется. Все выходные данные записываются в конец файла, а указатель файла устанавливается на конец выведенных данных. Если два отдельных процесса открывают один и тот же файл для пополнения, каждый процесс может записывать свои данные в файл без риска уничтожения данных, записываемых другим процессом. Данные, выводимые двумя процессами, будут перемешаны в файле, порядок их расположения в файле определяется порядком, в котором они записывались в файл.

FREAD, FWRITE(3S)

(ВВОД И ВЫВОД ДАННЫХ В ДВОИЧНОМ ПРЕДСТАВЛЕНИИ)

Ввод и вывод данных в двоичном представлении выполняются с помощью функций `fread` и `fwrite`. Эти функции имеют следующие спецификации:

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

```
int fread(ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;
```

```
int fwrite(ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;
```

Функция `fread` копирует `nitems` элементов из потока с именем `stream` в массив, начало которого определяется указателем `ptr`; каждый элемент данных является последовательностью байтов (эту последовательность нет необходимости завершать нулевым байтом) длиной `size`. Пересылка байтов при вызове функции `fread` прекращается, если встретился конец файла или возникла ошибка во время чтения данных из потока `stream`, или если уже считано `nitems` элементов. Функция `fread` оставляет

указатель файла (если он определен), установленный на байт потока stream, непосредственно следующий за последним считанным байтом (если он существует). Функция fread не изменяет содержимого потока stream.

Функция fwrite добавляет не более nitems элементов данных в выходной поток, выбирая эти данные из массива с указателем ptr. Добавление данных заканчивается при записи nitems элементов данных или возникновении ошибки. Содержимого массива с указателем ptr функция fwrite не изменяет.

Значение переменной size определяется обычно как sizeof(*ptr), где псевдофункция sizeof указывает длину элемента, на который установлен указатель ptr. Если указатель ptr обеспечивает ссылку на тип данных, отличный от типа char, то он должен быть преобразован в указатель на тип char.

FSEEK, REWIND, FTELL(3S) (установка указателя файла в потоке)

Функции fseek, rewind и ftell используются для потоков с прямым доступом. Они имеют следующие спецификации:

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

```
int fseek(stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;
```

```
void rewind(stream)
FILE *stream;
```

```
long ftell(stream)
FILE *stream;
```

С помощью функции fseek указывается расположение данных в потоке stream для следующей операции ввода или вывода. Положение данных указывается заданием смещения offset (с учетом его знака) относительно начала или текущего положения, или конца файла в зависимости от значения переменной ptrname (0, 1 или 2). Вызов функции rewind(stream) по своему действию эквивалентен вызову функции fseek(stream, 0L, 0), однако первая функция не возвращает в вызывающую программу никакого значения.

Функции fseek и rewind уничтожают все результаты, полученные функцией ungetc.

После вызова функции fseek или rewind следующей операцией с файлом, открытым для модификации, может быть операция либо ввода, либо вывода.

Функция `ftell` возвращает смещение обрабатываемого в данный момент байта относительно начала файла, ассоциированного с потоком `stream`.

GETC, GETCHAR, FGETC, GETW(3S) (считывание знака или слова из потока)

Макросы `getc` и `getchar` и функции `fgetc` и `getw` служат для считывания знаков и слов из потоков; их спецификации имеют следующий вид:

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

```
int getc(stream)
FILE *stream;
```

```
int getchar()
```

```
int fgetc(stream)
FILE *stream;
```

```
int getw(stream)
FILE *stream;
```

Макрос `getc` возвращает следующий знак (т.е. байт), полученный из входного потока `stream`, и передвигает указатель файла, если он определен, на один знак вперед. Поскольку `getc` является макросом, он не может использоваться в ситуациях, когда необходима функция; например, с ним нельзя использовать указатель на функцию, указывающий на него.

Макрос `getchar` возвращает следующий знак, полученный из стандартного входного потока `stdin`.

Функция `fgetc` по своему действию эквивалентна макросу `getc`, однако является настоящей функцией. Она работает медленнее, чем макрос `getc`, но на каждый вызов требует меньше памяти.

Функция `getw` возвращает следующее слово (т.е. значение типа `integer`), полученное из входного потока `stream`. Размер слова зависит от машины. В случае ошибки или конца файла функция `getw` возвращает константу EOF. Функция `getw` увеличивает значение соответствующего указателя файла, если он определен, для указания на следующее слово. Функция `getw` не предполагает, что в файле делается специальное выравнивание по границе слов.

GETS, FGETS(3S) (получение знаковой строки из потока)

Функции `gets` и `fgets` используются для считывания строк из входных потоков. Они имеют следующие спецификации:

```
#include <stdio.h>

char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
int n;
FILE *stream;
```

Функция `gets` считывает знаки из стандартного входного потока `stdin` и помещает их в массив, для которого параметр `s` является указателем. Считывание выполняется до тех пор, пока не будет считан знак перехода на новую строку или не встретится конец файла. Знак перехода на новую строку игнорируется, а строка завершается нулевым знаком.

Функция `fgets` считывает знаки из входного потока `stream` в массив, для которого параметр `s` является указателем. Считывание выполняется до тех пор, пока не будет считано `n-1` знаков или не будет считан знак перехода на новую строку, или не встретится конец файла. Считанная из входного потока строка завершается нулевым знаком.

MALLOC, FREE, REALLOC, CALLOC(3) (управление распределением основной памяти)

Функции `malloc` и `free` составляют простой пакет программ общего назначения для управления распределением памяти. Они имеют следующие спецификации:

```
char *malloc(size)
unsigned size;

void free(ptr)
char *ptr;
```

Функция `malloc` возвращает значение указателя на блок памяти размером не менее `size` байтов, расположение которого в памяти выбирается так, чтобы он был пригоден для любого использования.

Аргументом функции `free` является значение указателя на блок памяти, ранее выделенный функцией `malloc`; после завер-

шения работы функции `free` освободившаяся память доступна для распределения, содержимое памяти остается нетронутым.

Если пространство памяти, распределенное с помощью функции `malloc`, было перераспределено или функции `free` в качестве параметра передано случайное число, то результаты работы функции `free` непредсказуемы.

Функция `malloc` выделяет непрерывную область памяти достаточного объема, которую ей удастся найти при циклическом поиске, начинающемся с последнего выделенного или освобожденного блока памяти.

Функция `realloc` изменяет размер блока памяти, указанного значением `ptr`, до размера `size` байтов и возвращает значение указателя на этот (возможно, перемещенный на другое место) блок. Содержимое памяти в меньшей из двух областей памяти - новой и старой - не изменяется. Если в свободной области памяти недостаточно места для выделения блока размером `size` байтов, то функция `realloc` обращается к функции `malloc`, чтобы расширить свободную область памяти до `size` байтов, а затем перемещает данные на новое место. Функция `realloc` имеет следующую спецификацию:

```
char *realloc(ptr, size)
char *ptr;
unsigned size;
```

Функция `realloc` работает также в том случае, когда указатель `ptr` адресует блок памяти, освобожденный в результате последнего вызова функции `malloc`, `realloc` или `calloc`. Таким образом, последовательности вызовов функций `free`, `malloc` и `realloc` могут использовать стратегию поиска, используемую функцией `malloc`, для уменьшения фрагментации памяти.

Функция `calloc` служит для выделения и обнуления пространства памяти для массива из `nelem` элементов размером `elsize`. Функция `calloc` имеет следующую спецификацию:

```
char *calloc(nelem, elsize)
unsigned nelem, elsize;
```

Каждая из рассмотренных выше функций возвращает значение указателя на соответствующим образом выравненную область памяти (после допустимой корректировки значения указателя) для хранения объектов любого типа.

Если требуемая память недоступна, то функции `malloc`, `realloc` и `calloc` возвращают указатель `NULL`.

POPEN, PCLOSE(3S) (создание канала для процесса)

Функция `popen` создает канал между вызывающей программой и выполняемой командой. Аргументами функции `popen` являются указатели на строки, оканчивающиеся нулевым знаком и содержащие, соответственно, командную строку и указание режима ввода-вывода (`r` - для чтения, `w` - для записи). Возвращаемое значение является указателем потока, таким, который позволяет передавать данные, как стандартные входные данные команды, записывая их в файл `stream` (если режим ввода-вывода задан знаком `w`). Аналогично для режима, заданного знаком `r`. Функция `popen` имеет следующую спецификацию:

```
#include <stdio.h>

FILE *popen(command, type)
char *command, *type;
```

Поток, открытый с помощью функции `popen`, должен быть закрыт с помощью функции `pclose`, которая ждет завершения соответствующего процесса и возвращает код завершения команды. Функция `pclose` имеет следующую спецификацию:

```
#include <stdio.h>

int pclose(stream)
FILE *stream;
```

Поскольку открытые файлы разделяются процессами, то команда типа `r` может использоваться как фильтр ввода, а команда типа `w` - как фильтр вывода.

PRINTF, FPRINTF, SPRINTF(3S) (вывод данных с форматированием)

Для вывода данных с форматированием используются функции `printf`, `fprintf` и `sprintf`, имеющие следующие спецификации:

```
#include <stdio.h>

int printf(format [, arg] ... )
char *format;

int fprintf(stream, format [, arg] ... )
FILE *stream;
```

```
char *format;
```

```
int sprintf(s, format [, arg] ... )
```

```
char *s, format;
```

Функция `printf` помещает выводимые данные в поток стандартного вывода `stdout`, а функция `fprintf` - в поименованный поток вывода `stream`. Функция `sprintf` помещает "выводимые" данные, заканчивающиеся нулевым знаком `\0`, в последовательные байты, начиная с байта `*s`; предполагается, что пользователь сам должен позаботиться о достаточном объеме памяти для "выводимых" данных. Каждая функция возвращает число переданных знаков (для функции `sprintf` в число переданных знаков не включается знак `\0`) или отрицательное число, если при выводе произошла ошибка.

Преобразование, форматирование и вывод данных, задаваемых фактическими параметрами (или аргументами) `arg`, выполняется каждой функцией под управлением формата `format`. Формат представляет собой знаковую строку, содержащую объекты двух типов: обычные знаки, которые просто копируются в выходной поток, и спецификации формата (или спецификации преобразования), каждая из которых обеспечивает вывод одного или нескольких элементов данных, заданных аргументами `arg`. Если аргументов недостаточно для всех спецификаций формата, то результат выполнения функции не определен. Если аргументов задано больше, чем существует спецификаций формата, то лишние аргументы просто игнорируются.

Каждая спецификация формата начинается со знака `%`, за которым в приведенной ниже последовательности располагаются следующие поля спецификации:

1. Нуль или более флагов, изменяющих действие этой спецификации преобразования.
2. Необязательная строка из десятичных цифр, задающая минимальную ширину поля. Если для представления преобразованного значения требуется знаков меньше, чем позволяет ширина поля, то в поле добавляются знаки-заполнители слева (или справа, если установлен флаг выравнивания значения по левому краю поля (см. ниже)).
3. Указатель точности, который определяет минимальное количество выводимых цифр числа для преобразований, обозначаемых знаками `d`, `o`, `u`, `x` или `X`, количество цифр числа, выводимых после десятичной точки для преобразований, обозначаемых знаками `e` и `f`, максимальное количество знаков, которое будет выведено в соответствии с преобразованием, обозначаемым знаком `s`. Точность задается точкой, за которой записыва-

ется строка десятичных цифр; пустая строка интерпретируется как нуль.

4. Необязательная буква l, указывающая, что следующие за ней знаки d, o, u, x или X, обозначающие тип преобразования, относятся к параметру arg типа long integer.

5. Знак, обозначающий тип преобразования, которое надо выполнить над соответствующим параметром.

Ширина поля и точность могут указываться звездочкой (*), а не только строкой цифр. В этом случае ширина поля и точность определяются по целому значению некоторого параметра arg. Значение параметра arg, которое фактически преобразуется, не обрабатывается до тех пор, пока не встретится буква, обозначающая соответствующее преобразование. Таким образом, параметры arg, указывающие ширину поля или точность, должны встретиться в списке параметров раньше, чем параметр (если он имеется), для которого они используются при преобразовании.

Для установки флагов используются следующие знаки:

- Результат преобразования будет выровнен по левому краю поля
- + результат преобразования со знаком будет всегда начинаться со знака (- или +).
- Пробел Если первый символ преобразования со знаком не является знаком, то перед результатом будет выводиться пробел. Следовательно, если для преобразования заданы одновременно флаги Пробел и +, то флаг Пробел игнорируется.
- # Этот флаг указывает, что соответствующее значение при выводе должно быть представлено в альтернативной форме.
На преобразования с, d, s и u этот флаг не влияет. Для преобразования o он увеличивает точность так, чтобы первая цифра результата стала нулем. Для преобразования x (X) ненулевой результат будет иметь префикс 0x (0X). Для преобразований e, E, f, g и G результат всегда будет содержать десятичную точку, даже если после точки не будет ни одной цифры (обычно десятичная точка появляется в результате только тогда, когда после точки должна стоять хотя бы одна цифра). Для преобразований g и G, как обычно, нули после точки не удаляются.

Следующие знаки соответствуют спецификациям преобразований:

- d, o, u, x, X Целое значение параметра `arg` преобразуется в десятичное со знаком, беззнаковое восьмеричное, десятичное или шестнадцатеричное соответственно; буквы `abcdef` используются для преобразования `x`, а буквы `ABCDEF` - для преобразования `X`. Заданная при выводе точность определяет количество выводимых цифр; если преобразуемое значение представляется меньшим количеством цифр, то добавляются ведущие нули. Точность, предполагаемая по умолчанию равна 1. Результатом преобразования нулевого значения с точностью 0 является пустая строка.
- f Значения параметров `arg` типа `float` или `double` преобразуются в запись вида `[-]ddd.ddd`, где количество цифр после десятичной точки соответствует заданной точности. Если точность не указана, то предполагается 6 цифр; если задана нулевая точность, то в результате отсутствует десятичная точка.
- e, E Значения параметров `arg` типа `float` или `double` представляются записью вида `[-]d.ddde+dd`, где до десятичной точки стоит одна цифра, а количество цифр после десятичной точки соответствует заданной точности. Если точность не указана, то предполагается шесть цифр; если задана нулевая точность, то в результате отсутствует десятичная точка. Экспонента всегда содержит не менее двух цифр. Если в спецификации формата указана буква `E` вместо `e`, то такая же буква будет выведена в представлении числа.
- g, G Значения параметров `arg` представляются так же, как при преобразовании `f` или `e` (или, если указана буква `G`, как при преобразовании `G`), количество значащих цифр определяется заданной точностью. Представление результата имеет такой же вид, как и при преобразовании `e`, если значение экспоненты после преобразования меньше -4 или больше, чем заданная точность. Незначащие нули после десятичной точки удаляются из результата, десятичная точка в результате выводится только в том случае, когда после нее следует цифра или установлен флаг `#`.
- c Выводится параметр `arg` типа `character`.

- s Параметр `arg` рассматривается как строка (знаковый указатель), знаки которой выводятся до обнаружения знака `\0` или до достижения количества выведенных знаков, определенного спецификацией точности. Если точность не указана, то она предполагается бесконечной и вывод знаков продолжается до первого встреченного знака `\0`. Если параметр `arg` является указателем на строку и имеет нулевое значение, то результат не определен. Результат не определен также, если выводится пустая строка.
- % Печатается знак %, аргумент не преобразуется.

При неправильном задании ширины поля или при заведомо малой его ширине поле игнорируется. Если результат преобразования не умещается в заданное поле, оно просто расширяется, чтобы полученный результат мог быть в нем размещен. Знаки, генерируемые функциями `printf` и `fprintf` выводятся так, как если бы была вызвана функция `putc`.

Ниже приведены два примера использования функции `printf`:
Печать даты и времени в форме

```
Sunday, July 3, 10:02
```

можно осуществить с помощью оператора

```
printf("%s, %s %d, %.2d:%.2d", weekday,
      month, day, hour, min);
```

Число "пи" с пятью знаками после десятичной точки можно напечатать с помощью оператора

```
printf("pi = %.5f", 4*atan(1.0));
```

PUTC, PUTCHAR, FPUTC, PUTW(3S)
(вывод знака или слова в поток вывода)

Макросы `putc` и `putchar` и функции `fputc` и `putw` используются для вывода знака или слова. Они имеют следующие спецификации:

```
#include <stdio.h>

int putc(c, stream)
char c;
FILE *stream;
```

```

int putchar(c)
char c;

int fputc(c, stream)
char c;
FILE *stream;

int putw(w, stream)
int w;
FILE *stream;

```

Макрос `putc` записывает знак `c` в выходной поток `stream` (место записи задается указателем файла, если он определен). Макрос `putchar(c)` определяется как `putc(c, stdout)`.

Поведение функции `fputc` аналогично поведению макроса `putc`, однако она является функцией, а не макросом. Для выполнения функции `fputc` требуется больше времени, но ее использование позволяет экономить память.

Функция `putw` записывает в выходной поток `stream` слово `w` (место записи задается указателем файла, если он определен), т.е. данные, соответствующие переменной типа `integer`, объем которых зависит от типа ЭВМ. При записи не предполагается выравнивания по какой-либо границе в файле, и такое выравнивание не выполняется.

Если при выводе есть обращение к файлу, то для выходных потоков, за исключением стандартного потока ошибок `stderr`, по умолчанию выполняется буферизация, при обращении к терминалу выполняется строчная буферизация.

Для стандартного выходного потока ошибок `stderr` по умолчанию буферизация не выполняется, однако использование функции `fopen` (см. описание функции `fopen`) дает возможность организовать при выводе буферизацию или строчную буферизацию. При отсутствии буферизации информация, предназначенная для записи в файл или вывода на терминал, выстраивается в очередь; при работе с буфером выводимые знаки предварительно накапливаются в буфере и выводятся блоками; при строчной буферизации строки знаков выстраиваются в очередь для вывода на терминал, очередная строка выводится, как только завершается передача данных этой строки (т.е. как только передан знак перехода на новую строку или запрошен ввод с терминала).

PUTS, FPUTS(3S) (вывод строки в поток вывода)

Функция `puts` обеспечивает вывод строки, заканчивающейся нулевым знаком, в стандартный выходной поток `stdout`. После

строки выводится знак перехода на новую строку. Параметр `s` является указателем на строку. Функция `puts` имеет следующую спецификацию:

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

```
int puts(s)
char *s;
```

Функция `fputs` обеспечивает вывод строки, заканчивающейся нулевым знаком, в выходной поток `stream` и имеет следующую спецификацию:

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

```
int fputs(s, stream)
char *s;
FILE *stream;
```

Ни одна из функций не выводит нулевого знака. В случае ошибки обе функции возвращают код EOF.

QSORT(3) (быстрая сортировка)

Функция `qsort` является реализацией алгоритма быстрой сортировки, в процессе которой данные остаются в одной и той же области памяти. Функция `qsort` имеет следующую спецификацию:

```
void qsort((char *) base, nel, sizeof(*base), compar)
unsigned int nel;
int (*compar)(.);
```

Аргумент `base` указывает элемент таблицы, расположенный у ее базового адреса. Аргумент `compar` является именем функции сравнения, вызываемой с двумя аргументами, указывающими на сравниваемые элементы. Функция должна вернуть одно из значений: отрицательное, равное нулю или положительное, в зависимости от того, меньше, равен или больше второго первый аргумент .

SCANF, FSCANF, SSCANF(3S) (ввод данных по формату)

Функции `scanf`, `fscanf` и `sscanf` используются для ввода данных по формату и имеют следующие спецификации:


```
#include <stdio.h>

int scanf(format [, pointer ] ...)
char *format;

int fscanf(stream, format [, pointer] ...)
FILE *stream;
char *format;

int sscanf(s, format [, pointer] ...)
char *s, *format;
```

Функция `scanf` служит для чтения данных из стандартного входного потока `stdin`. Функция `fscanf` служит для чтения данных из входного потока, имя которого указывается параметром `stream`. Функция `sscanf` читает данные из строки знаков `s`. Каждая функция обеспечивает чтение знаков, их интерпретацию в соответствии с форматом и запись результатов в указанные аргументами функции области памяти. Для каждой функции в качестве аргументов необходимо задать описываемую ниже управляющую строку `format` и набор указателей `pointer`, обозначающих, куда должны быть записаны преобразованные при вводе данные.

Управляющая строка обычно содержит спецификации преобразования, которые используются для интерпретации входных последовательностей знаков. Управляющая строка может содержать следующие знаки и спецификации:

1. Знаки пробела, табуляции, перехода на новую строку или продвижения страницы, которые, за исключением двух описанных ниже случаев, обеспечивают чтение входных данных до ближайшего знака, отличного от перечисленных выше знаков.

2. Обычный знак (отличный от `%`), который должен соответствовать следующему знаку входного потока.

3. Спецификации преобразований, состоящие из знака `%`, обязательного знака запрета присваивания `*`, необязательного указания максимальной ширины числового поля, необязательных знаков `l` или `h`, указывающих число байт, соответствующих принимающей переменной, и кода преобразования.

Спецификация преобразования управляет преобразованием следующего поля входных данных; результат присваивается переменной, указываемой соответствующим аргументом, если присваивание не запрещено присутствием знака `*`. Запрет присваивания обеспечивает возможность описания поля входных данных, которое должно быть пропущено при вводе. Поле входных данных определяется как строка знаков, отличных от пробела; граница поля определяется первым встреченным недопус-

тимым внутри поля знаком или превышением значения ширины поля, если она задана.

Код преобразования указывает, как интерпретировать поле входных данных; отвечающий ему указатель-аргумент обычно должен иметь соответствующий тип. Для поля с запрещенным присваиванием данных задавать указатель нельзя. Допускаются следующие коды преобразования:

| | |
|-------|---|
| % | В качестве элемента входных данных ожидается один знак %, присваивания значения не выполняется. |
| d | В качестве элемента входных данных ожидается десятичное целое; соответствующий аргумент должен быть указателем на целое. |
| u | В качестве элемента входных данных ожидается беззнаковое десятичное целое; соответствующий аргумент должен быть указателем на беззнаковое целое. |
| o | В качестве элемента входных данных ожидается восьмеричное целое; соответствующий аргумент должен быть указателем на целое. |
| x | В качестве элемента входных данных ожидается шестнадцатеричное целое; соответствующий аргумент должен быть указателем на целое. |
| e,f,g | В качестве элемента входных данных ожидается число с плавающей точкой; после преобразования данные записываются в память по адресу, указанному соответствующим аргументом; этот аргумент должен быть указателем на число с плавающей точкой. Входной формат для чисел с плавающей точкой задается строкой цифр со знаком или без знака и может включать в себя десятичную точку с полем экспоненты, состоящим из буквы E или e и следующим за ней целым числом (со знаком или без знака). |
| s | В качестве элемента входных данных ожидается строка знаков; соответствующий аргумент должен быть указателем на знак, указывающим на массив знаков, места в котором достаточно для размещения всей строки вместе с конечным знаком \0, добавляемым автоматически. Поле входных данных завершается знаком пробела, табуляции, перехода на новую строку или перехода на новую страницу. |
| c | В качестве элемента входных данных ожидается некоторый знак; соответствующий аргумент должен быть указателем на знак. В этом слу- |

чае пробелы и другие разделители не выполняют своих обычных функций; для чтения ближайшего знака, не являющегося пробелом, нужно использовать спецификацию `%s`. Если задана ширина поля, то соответствующий аргумент относится к массиву знаков, при этом читается указанное число знаков.

[Этот знак указывает на ввод строки данных, для которой разделители (пробел и т.п.) не выполняют своих обычных функций. После левой квадратной скобки записывается набор знаков, называемых знаками сканирования, после которых ставится правая квадратная скобка. Соответствующее этой спецификации входное поле представляет собой последовательность входных знаков максимальной длины, состоящую целиком из знаков сканирования. Знак `^`, если он является первым знаком в наборе знаков сканирования, играет роль оператора дополнения и переопределяет набор знаков сканирования как набор всех знаков, не содержащихся в строке знаков сканирования. Относительно набора знаков сканирования имеется ряд соглашений. Интервал знаков можно представить конструкцией "первый-последний", например, последовательность `[0123456789]` можно представить, как `[0-9]`. При использовании этого соглашения необходимо следить, чтобы первый знак интервала был лексикографически меньше, чем последний знак интервала, в противном случае знак минус в записи интервала просто изображает сам себя и не задает интервала знаков. Знак минус интерпретируется как знак минус и в том случае, когда он является первым или последним знаком среди набора знаков сканирования. Для включения знака первой квадратной скобки в набор знаков сканирования необходимо его поставить в наборе на первое место (или сразу после знака `^`, если он также включен в набор), тогда он не будет интерпретироваться как ошибка, указывающая конец списка знаков сканирования. Соответствующий аргумент должен указывать на знаковый массив, обеспечивающий размещение всех вводимых данных, включая знак `\0`, который при вводе добавляется автоматически.

Перед знаками d, u, o, и x, задающими преобразование данных при вводе, может стоять буква l или h, обозначающая, что в списке аргументов должен быть соответствующий указатель на переменную типа long или short, а не на переменную типа int. Аналогично, буква l, стоящая перед знаками e, f и g в спецификациях преобразований, требует аргумента-указателя на переменную типа double, а не float.

Преобразование, выполняемое функцией scanf, заканчивается при появлении кода EOF, при достижении конца управляющей строки или несоответствия знака входных данных управляющей строке. В последнем случае знак-нарушитель не считывается из входного потока.

После завершения функция scanf возвращает число элементов входных данных, значения которых присвоены переменным в соответствии со спецификациями преобразований. Это число может быть равным нулю, если несоответствие знака входных данных и управляющей строки обнаружилось в самом начале ввода. Если входные данные закончились до первого несоответствия или преобразования, то возвращается код EOF. Ниже приведено несколько примеров использования функции scanf:

1. При использовании определений

```
int i; float x; char name[50];
```

вызов функции scanf вида

```
scanf("%d%f%s", &x, name);
```

при строке входных данных

```
25 54.32E-1 thompson
```

обеспечит присваивание значения 25 переменной i, значения 54.32 переменной x и значения thompson\0 переменной name.

2. При использовании определения переменных, приведенных выше, вызов функции scanf вида

```
scanf("%2d%f%d %[0-9]", ^i, ^x, name);
```

при строке входных данных

```
56789 0123 56a72
```

обеспечит присваивание значения 56 переменной i, значения 789.0 переменной x и поместит строку 56\0 в область памяти с

именем `name`. Последующее обращение к функции `getchar` (см. описание функции `getc`) обеспечит возврат значения `a`.

SETJMP, LONGJMP(3) (нелокальный переход)

Функции `setjmp` и `longjmp` оказываются полезными при обработке ошибок и прерываний в подпрограммах нижнего уровня. Эти функции имеют следующие спецификации:

```
#include <setjmp.h>

int setjmp(env)
jmp_buf env;

void longjmp(env, val)
jmp_buf env;
int val;
```

Функция `setjmp` сохраняет состояние своего стека в области памяти с именем `env` (тип `jmp_buf` переменной `env` определен в файле `<setjmp.h>`), которая затем используется функцией `longjmp`. Функция `setjmp` возвращает значение 0.

Функция `longjmp` восстанавливает состояние среды, сохраненное последним вызовом функции `setjmp` с помощью соответствующего аргумента `env`. После завершения работы функции `longjmp` выполнение программы продолжается, как если бы соответствующий вызов функции `setjmp` возвращал значение `val`. Функция `longjmp` не может повлиять на возврат функцией `setjmp` значения 0. Если функция `longjmp` вызывается со значением второго аргумента, равным 0, то функция `setjmp` возвращает значение 1. Значения всех данных, к которым возможен доступ, остаются такими, которые были в момент вызова функции `longjmp`.

SLEEP(3)

(приостановка выполнения программы на заданное время)

Функция `sleep`, описываемая как

```
unsigned sleep(seconds)
unsigned seconds;
```

приостанавливает выполнение текущего процесса на время, указываемое аргументом функции. Время задается в секундах.

Фактическое время приостановки выполнения процесса может быть меньше, чем запрошенное, по следующим двум причинам:

1. Запланированное продолжение выполнения процесса попадает в односекундный интервал.

2. Некоторый принятый сигнал завершает выполнение функции sleep вследствие выполнения соответствующей программы обработки сигнала.

Время приостановки может оказаться большим запрошенного на произвольное число секунд вследствие других событий в системе. Значение, возвращаемое функцией sleep, представляет собой разность между запрошенным временем и временем фактической приостановки выполнения.

Процедура приостановки выполнения реализуется установкой предупреждающего сигнала с последующим ожиданием этого (или какого-либо другого) сигнала. Предыдущее состояние предупреждающего сигнала сохраняется и восстанавливается. В вызывающей программе предупреждающий сигнал может быть установлен до вызова функции sleep; если время приостановки выполнения процесса, заданное функцией sleep, превышает время, оставшееся до такого предупреждающего сигнала, то процесс возобновляется с момента, когда должен быть подан предупреждающий сигнал, а программа, которой передается управление вызывающей программой по предупреждающему сигналу, выполняется непосредственно перед возвратом из функции sleep. Если же время приостановки процесса меньше, чем время, оставшееся до предупреждающего сигнала, то предупреждающий сигнал подается через такое время, как если бы не было вызова функции sleep.

STRCAT, STRNCAT, STRCMP, STRNCMP,
STRCPY, STRNCPY, STRLEN, STRCHR, STRRCHR,
STRPBRK, STRSPN, STRCSPN, STRTOK(3)
(операции над строками)

Пречисленные в заголовке раздела функции предназначены для обработки строк, заканчивающихся нулевым знаком, и имеют следующие спецификации:

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

```
char *strcat(s1, s2)  
char *s1, *s2;
```

```
char *strncat(s1, s2, n)  
char *s1, *s2;  
int n;
```

```
int strcmp(s1, s2)  
char *s1, *s2;
```

```

int strncmp(s1, s2, n)
char *s1, *s2;
int n;

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;
int n;

int strlen(s)
char *s;

char *strchr(s, c)
char *s, c;

char *strrchr(s, c)
char *s, c;

char *strpbrk(s1, s2)
char *s1, *s2;

int strspn(s1, s2)
char *s1, s2;

int strcspn(s1, s2)
char *s1, *s2;

char *strtok(s1, s2)
char *s1, *s2;

```

Аргументы *s1*, *s2* и *s* указывают на строки (массивы знаков, оканчивающиеся нулевым знаком). Функции *strcat*, *strncat*, *strcpy* и *strncpy* изменяют значение аргумента *s1*. Эти функции не проверяют на переполнение массивы, соответствующие указателю *s1*.

Функция *strcat* добавляет копию строки *s2* к концу строки *s1*. Функция *strncat* добавляет не более *n* знаков. Каждая из функций возвращает указатель на результат, т.е. на строку, оканчивающуюся нулевым знаком.

Функция *strcmp* сравнивает значения аргументов и возвращает целое значение меньше, равное или больше нуля в зависимости от того, лексикографически меньше, равна или больше строки *s2* строка *s1*. Функция *strncmp* выполняет такое же сравнение, но для сравнения используется только *n* знаков.

Функция `strcpy` копирует строку `s2` в строку `s1`, операция копирования заканчивается, если встречен нулевой знак. Функция `strncpy` копирует точно `n` знаков, перенося не все знаки строки `s2` или добавляя к строке `s1` нулевые знаки, если это необходимо. Если длина строки `s2` больше или равна `n`, результирующая строка не заканчивается нулевым знаком. Обе функции возвращают `s1`.

Функция `strlen` возвращает число знаков в строке `s`; нулевой знак, завершающий строку, не включается в это число.

Функция `strchr` (`strrchr`) возвращает указатель на первый (последний) знак `c`, найденный в строке `s`, или указатель `NULL`, если знак `c` не найден в строке. Нулевой знак, завершающий строку, считается частью строки.

Функция `strpbrk` возвращает указатель на первый встретившийся в строке `s1` знак строки `s2` или указатель `NULL`, если в строке `s1` не встретилось ни одного знака строки `s2`.

Функция `strspn` (`strcspn`) возвращает значение длины начального сегмента строки `s1`, целиком состоящего из знаков строки `s2` (целиком состоящего из знаков, отсутствующих в строке `s2`).

Функция `strtok` рассматривает строку `s1` как последовательность текстовых элементов (их число больше или равно нулю), разделенных одним или более знаками из строки `s2`, выступающей в роли источника разделителей. Первый вызов функции (с заданным значением указателя `s1`) возвращает указатель на первый знак первого элемента и записывает нулевой знак в строку `s1` сразу после этого элемента. Функция хранит информацию, необходимую для последующих вызовов (которые должны выполняться с указателем `NULL`), обеспечивающих обработку строки `s1` со знака, непосредственно следующего за элементом строки, обработанным во время предыдущего вызова функции. Строка-разделитель `s2` может меняться от вызова к вызову. После того как исчерпаны все элементы строки `s1`, функция возвращает указатель `NULL`.

Для удобства пользователей все перечисленные выше функции описаны в необязательном файле-заголовке `<string.h>`.

TMPFILE(3S) (создание временного файла)

Функция `tmpfile` создает временный файл и возвращает соответствующий указатель `FILE`. Эта функция имеет следующую спецификацию:

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

```
FILE *tmpfile()
```


Образованный с помощью функции `tmpfile` файл автоматически удаляется при завершении использующего его процесса. Этот файл открывается для корректировки.

TMPNAM, TEMPNAM(3S) (присваивание имени временному файлу)

Функции `tmpnam` и `tempnam` служат для генерации имен временных файлов с выбором имен, не создающих конфликтов в системе. При каждом вызове эти функции генерируют различные имена, спецификации функций имеют следующий вид:

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

```
char *tmpnam(s)  
char *s
```

```
char *tempnam(dir, pfx)  
char *dir, *pfx;
```

Функция `tmpnam` всегда генерирует имя файла, используя имя пути, определенное в файле `stdio.h` как `P_tmpdir`. Если значение переменной `s` равно значению `NULL`, функция `tmpnam` оставляет полученный результат во внутренней статической области и возвращает указатель на эту область. Следующий вызов функции `tmpnam` разрушает содержимое этой области. Если значение переменной `s` не равно значению `NULL`, то предполагается, что оно является адресом массива из не менее чем `L_tmpnam` байтов, а константа `L_tmpnam` определяется в файле `stdio.h`; функция `tmpnam` помещает результат в этот массив и возвращает значение `s`.

Функция `tempnam` позволяет пользователю управлять выбором каталога. Аргумент `dir` указывает на путь к каталогу, в котором должен быть создан файл. Если `dir` имеет значение `NULL` или указывает на строку, не являющуюся именем пути к соответствующему каталогу, то используется имя пути, определенное в файле `stdio.h` как `P_tmpdir`. Если же это имя пути недоступно, то в качестве последнего средства используется имя `/tmp`. Можно также в среде пользователя ввести переменную среды `TMPDIR`, значением которой будет имя пути для желаемого каталога с временным файлом.

Во многих приложениях принято соглашение, в соответствии с которым имена файлов должны начинаться с определенной последовательности букв. Такое требование можно выполнить, используя аргумент `pfx`. Этот аргумент может иметь значение `NULL` или указывать на строку из не более чем пяти знаков, используемых в качестве первых знаков имени временного файла.

Для резервирования области памяти, необходимой для конструируемого имени временного файла, в функции `tempnam` используется функция `malloc`. При выполнении функции возвращается указатель на эту область. Таким образом, значение указателя, возвращенное функцией `tempnam`, может служить аргументом функции `free` (см. описание функции `malloc`). Если по какой-либо причине (например, при неудачном завершении выполнения функции `malloc` или при безуспешности вышеупомянутых попыток найти соответствующий каталог) функция `tempnam` не может вернуть в вызывающую программу ожидаемый результат, то возвращается указатель `NULL`.

UNGETC(3S) (возврат знака во входной поток)

Функция `ungetc` имеет следующую спецификацию:

```
#include <stdio.h>

int ungetc(c, stream)
char c;
FILE *stream;
```

Функция `ungetc` помещает знак `c` в буфер, ассоциированный с входным потоком `stream`. При завершении работы функция `ungetc` возвращает знак `c` и оставляет файл `stream` без изменения.

Возврат одного знака во входной поток гарантируется при условии, что из потока `stream` ранее было что-то прочитано, а ввод из потока действительно выполнялся с буферизацией.

Если значение знака `c` равно значению `EOF`, то функция не выполняет с буфером никаких действий и возвращает `EOF`.

Функция `fseek` очищает всю память от возвращенных знаков.

A2. МАТЕМАТИЧЕСКАЯ БИБЛИОТЕКА `libm`

1. Функции Бесселя (`j0`, `j1`, `jn`, `y0`, `y1` и `yn`).
2. Функции обработки ошибок (`erf` и `erfc`).
3. Функции вычисления:
 - экспоненты (`exp`),
 - логарифма (`log` и `log10`),
 - степени (`pow`),
 - квадратного корня (`sqrt`).
4. Функции получения:
 - целой части числа (`floor`),
 - минимального целого числа, большего заданного (`ceil`),
 - остатка от деления (`fmod`),

- абсолютного значения (fabs).
5. Гамма-функция (gamma).
 6. Функция вычисления длины гипотенузы (hypot).
 7. Гиперболические функции (sinh, cosh и tanh).
 8. Тригонометрические функции (sin, cos, tan, asin, acos, atan и atan2).

Приложение Б

Некоторые инструментальные средства

Операционная система UNIX обеспечивает для программирования на языке Си широкий выбор инструментальных программ, облегчающих процесс программирования. Например, существуют средства для выявления ошибок в программах, компиляции, представления текстов программ в формате, удобном для чтения, а также средства, облегчающие сопровождение программ.

Использование этих инструментальных программ, за исключением компилятора, является необязательным; однако все больше и больше программистов используют их в своей работе, оценив простоту и эффективность их применения. Здесь будут описаны лишь некоторые из широко используемых инструментальных программ. Более полный список инструментальных средств содержится в [3-5].

Б1. ПРОВЕРОЧНАЯ ПРОГРАММА lint

С помощью программы lint можно выявить все те особенности программ на языке Си, которые чреваты ошибками или затрудняют перенос программ с одного компьютера на другой. Программа lint более строго, чем компилятор, проверяет программу на языке Си с целью выявления неправильного использования типов, она указывает на потенциальные источники ошибок, такие как недостижимые операторы, циклы, вход в которые осуществляется, минуя его начало, и автоматические переменные, определяемые, но не используемые в программе. С помощью программы lint проверяется также, имеют ли фактические параметры вызываемой функции соответствующие типы, правильно ли осуществляется возврат значений после завершения выполнения функций, вызываются ли они с переменным числом аргументов и используются ли возвращаемые функциями значения (т.е. не используются ли функции в качестве подпрограмм).

Обращение к программе lint имеет следующий вид:

lint [-abchnpvx] *file₁.c file₂.c ... file_n.c*

По умолчанию предполагается, что файлы *file_i* должны быть загружены все вместе; они проверяются на обоюдную совместимость. При обращении к программе может быть использовано любое число следующих опций:

- | | |
|---|---|
| h | Выполнить эвристические проверки для обнаружения ошибок, улучшения стиля и уменьшения избыточности. |
| b | Сообщить о недостижимых в процессе выполнения операторах break. |
| v | Запретить выдачу сообщений о неиспользуемых аргументах в функциях. |
| x | Сообщить о переменных, описанных как extern, но нигде не используемых. |
| a | Сообщить о присваиваниях значений типа long переменным типа int. |
| c | Сообщить о явных преобразованиях типов, возможность переноса которых на компьютеры других типов вызывает сомнения. |
| u | Не сообщать о функциях и переменных, используемых, но не определенных, или определенных, но не используемых (такая ситуация возможна при работе программы lint с подмножеством файлов для большой программы). |
| n | Не проверять совместимости со стандартной библиотекой. |

В процессе проверки программа lint не в состоянии правильно квалифицировать обращение к функции exit и другим функциям, не возвращающим значения, что приводит к ложным сообщениям об ошибках.

Приведенные ниже комментарии, будучи вставленными в программу на языке Си, меняют поведение программы lint следующим образом:

- | | |
|----------------|--|
| /*NOTREACHED*/ | В соответствующих точках программы не сообщается о недостижимых кодах. |
| /*VARARGSn*/ | Отменяется обычная проверка, является ли переменным число аргументов в следующем описании функции. Для первых n аргументов проверяются типы данных; опущенное n и n, равное 0, эквивалентны. |
| /*NOSTRICT*/ | Отменяется строгая проверка типов в следующем выражении. |
| /*ARGSUSED*/ | Для следующей функции включается опция -v. |

*/*LINTLIBRARY*/* В начале файла отменяется выдача сообщений о неиспользуемых функциях из числа имеющихся в этом файле.

В2. КОМПИАТОР сс ЯЗЫКА СИ

В системе UNIX компилятор языка Си [3] воспринимает аргументы нескольких типов. Аргументы с именами, оканчивающимися на точку и букву с, считаются исходными текстами программ на языке Си; после компиляции каждая объектная программа находится в файле с тем же самым именем, что и ее исходный текст, но знаки .с заменены теперь знаками .о.¹ Если отдельная программа на языке Си сразу и компилируется и загружается, тогда появляется файл a.out, а объектный файл удаляется.

В качестве других аргументов могут задаваться необязательные аргументы загрузчика; совместимые с программами на языке Си объектные модули, обычно полученные при предыдущих прогонах компилятора; библиотеки программ, совместимых с программами на языке Си. Эти программы вместе с результатами указанных компиляций загружаются (в указанном порядке) для получения выполняемой программы с именем a.out.

Вызов компилятора имеет следующий вид:

сс [опции] *file₁ file₂ ... file_n*

Опции компилятора задаются с помощью следующих ключей:

- | | |
|---------------|--|
| -с | Отменяет фазу загрузки при компиляции и требует создания объектного файла даже в том случае, когда компилируется одна программа. |
| -w | Отменяет выдачу диагностических сообщений. |
| -О | Вызывает оптимизацию объектного кода. |
| -Е | Для программ с указанными именами вызывает прогон только макропроцессора с пересылкой результатов в файл стандартного вывода. |
| -С | Запрещает макропроцессору исключать из текста программы комментарии. |
| -о <i>имя</i> | Вызывает присваивание выходному файлу имени <i>имя</i> . При использовании этой опции файл a.out остается нетронутым. |

¹ Более точно, в системе UNIX этот объектный файл (т.е. файл с суффиксом .о) помещается в текущий каталог.

Б3. ПРОГРАММА ФОРМАТИРОВАНИЯ cb

Программа форматирования cb в качестве входных данных получает исходный текст программы на языке Си, введенный из стандартного входного потока, и выдает в качестве результата текст программы с пробелами и отступами, делающими структуру программы более наглядной. Полученная таким образом программа записывается в стандартный выходной поток. Обращение к программе форматирования имеет следующий вид:

```
cb <fileinput >fileoutput
```

Рассмотрим следующую программу, написанную без отступов:

```
#include <stdio.h>
int x[3] = {1, 2, 3};
main()
{
void swap();
swap (&x[1], &x[2]);
printf( "%d", x[1]);
}
void swap(a, b)
int *a, *b;
{
int t;
t = *a;
*a = *b;
*b = t;
}
```

Программа форматирования преобразует этот текст к более удобному для чтения:

```
#include <stdio.h>
int x[3] = {
    1, 2, 3};
main()
{
    void swap();
    swap (&x[1], &x[2]);
    printf( "%d", x[1]);
}
void swap(a, b)
int *a, *b;
{
    int t;
```

```
t = *a;  
*a = *b;  
*b = t;  
}
```

Вы можете, конечно, предпочесть свой стиль размещения текста программы, а не пользоваться программой `sv`. Независимо от стандарта на размещение текста программы, которого придерживается программист, все усилия, предпринимаемые для облегчения чтения текста программы, делаются не столько для самого программиста, сколько для тех, кто будет пытаться понять работу этой программы.

Б4. ПРОГРАММА СОПРОВОЖДЕНИЯ ГРУППЫ ПРОГРАММ `make`

Преимущества разработки систем программного обеспечения на основе модульного подхода общепризнаны. Однако при реализации этого подхода возникает один практический вопрос. Если проведены изменения в некотором модуле (в файле с исходным текстом на языке Си), то как среди остальных модулей найти все те модули, которые зависят от этих изменений? Наиболее сложно ответить на этот вопрос на начальном этапе разработки программы, когда из-за коррекции обнаруженных ошибок, изменения спецификаций программы или нахождения лучшего решения изменения в модулях проводятся наиболее часто.

Существует два очевидных, но не лучших решения рассматриваемой задачи: компилировать все модули системы при каждом изменении в каком-либо модуле или компилировать часть модулей, зависящих от проведенного изменения. Первое решение требует больших дополнительных затрат ресурсов и длительного ожидания результата, так как часто необходимо компилировать лишь некоторые модули. Второе решение чревато ошибками, особенно при большом числе модулей и их частой модификации. Достаточно забыть о компиляции хотя бы одного модуля или нарушить правильную последовательность компиляции, чтобы получить неработоспособный вариант всей системы.

Поставленная задача решается автоматизацией процесса выборочной компиляции с помощью программы `make`; при этом программисту не нужно помнить, на какие модули влияет проведенное изменение и какова правильная последовательность компиляции этих модулей. Программа `make` использует предоставленную пользователем информацию о зависимости файлов и данные о времени последней модификации файла, получаемые из файловой системы. В программу `make` заложены также сведения об отношениях между некоторыми типами объ-

ектов. Например, программе `make` известно, что любой файл с суффиксом `.o` может быть получен компиляцией соответствующего файла с тем же префиксом и с суффиксом `.c`.

В качестве примера, иллюстрирующего использование программы `make`, рассмотрим программу сортировки `sort`, составленную из следующих пяти модулей:

| Модуль | Выполняемая функция |
|---------------------|---|
| <code>sort</code> | Содержит всю программу сортировки в выполняемой форме. |
| <code>main.c</code> | Содержит главную программу, которая считывает входные данные из стандартного входного потока и вызывает функцию <code>qksort</code> (содержащуюся в файле <code>qksort.c</code>) для сортировки массива введенных элементов, а затем записывает отсортированный массив в стандартный выходной поток. |
| <code>qksort</code> | Содержит функцию <code>qsort</code> , которая собственно и выполняет сортировку; эта функция обращается к функции <code>part</code> для разбиения массива и к функции <code>swap</code> для перестановки двух элементов массива. |
| <code>part.c</code> | Содержит функцию <code>part</code> . |
| <code>swap.c</code> | Содержит функцию <code>swap</code> . |
| <code>defs</code> | Содержит определения, используемые в программе <code>main.c</code> , которые включаются в нее предложением препроцессора <code>include</code> . |

Для описания действий, которые необходимо предпринять при модификации файлов, используются связи между файлами, задаваемые следующим образом:

```
sort:  main.o qksort.o part.o swap.o
      cc -o sort main.o qksort.o part.o swap.o
main.o: defs
```

В первой строке сообщается, что файл `sort` зависит от файлов `main.o`, `qksort.o`, `part.o` и `swap.o`, а в третьей строке - что файл `main.o` зависит от файла `defs`. Нет необходимости указывать связь между объектными файлами и соответствующими файлами с исходным текстом на языке Си. Как уже упоминалось, эта информация заранее заложена в программу `make`.

Информация о связях между файлами и соответствующих действиях помещается в файл с именем `makefile`. При любых

изменениях в модулях выполнением команды `make` обеспечивается компиляция всех необходимых файлов. Если все модули уже соответствуют современному состоянию программы, то программа `make` напечатает сообщение

```
'sort' is up to date
```

и закончит работу. Если для всех объектных файлов уже имеются последние версии, а новая версия программы `sort` еще не создана, то будут выполнены действия, предписанные второй строкой файла `makefile`:

```
cc -o sort main.o qksort.o part.o swap.o
```

Если для некоторого объектного модуля последней версии не существует, то такой модуль будет получен автоматически компиляцией соответствующего исходного текста (т.е. `main.c`, `qksort.c`, `part.c` и `swap.c`). Предположим, что изменения проведены в файле определений `defs`. Программа `sort` зависит от файла `main.o`, который, в свою очередь, зависит от файла `defs`. Следовательно, сначала будет перекомпилирован файл `main.o`, а затем будет выполнена вторая строка вышеприведенного файла зависимости для построения скорректированной версии программы `sort`.

В качестве другого примера рассмотрим сокращенный вариант файла `makefile`, используемый при разработке прототипа системы подготовки документации [27]:

```
#определения макросов имеют вид NAME = строка;
#вызов макроса имеет вид $(NAME)
#если текст, не являющийся комментарием, не уместается
#на строке, то его можно продолжить на следующей,
#заканчивая продолжаемую строку знаком обратной дробной
#черты

LINT =      lint

CFLAGS =    -O          #опция компилятора для
                        #получения оптимизированного кода

BEEP =      "^G"        #control G используется для
                        #воспроизведения звукового
                        #сигнала терминала

FILES =     Makefile    \
             createmask \
             enhncemsk.c \
             mask_pre.c  \
```

```

tra.disp.tbl \
help_pos.c   \
forms        \
emp.db       \
emp.dbrt.c   \
formI.c      \
str.h        \
tra.fld.defs \
tra.pre.c    \
tra.post.c   \
tra.acc_rts  \
acc_rts.c    \
tra.fld.types \
tra.error.c  \
tra.template \
tra.number   \
tra.actions.c \
hpsub.c

INTP =      formI.o \
            hpsub.o \
            tra.pre.o \
            tra.post.o \
            acc_rts.o \
            tra.error.o \
            tra.actions.o \
            emp.dbrt.o

MASK =      creatmask \
            enhncemsk \
            mask_pre

formI:      $(INTP) #макровывзов, дающий
#           список файлов, от которых
#           зависит файл formI
            $(CC) $(CFLAGS) -o formI $(INTP)
#           макровывзов $(CC) -о formI эквивалентен "cc";
#           загружает и связывает все файлы
#           списка, получаемого по $(INTP), в
#           выполняемый файл "formI"; CC -
#           заранее определенный макрос

formI.o tra.pre.o tra.post.o \
tra.error.o tra.actions.o:   tra.fld.defs

tra.mask:      $(MASK) tra.disp.tbl
               @echo "создается файл tra.mask"
#             знак @ обеспечивает вывод только
#             сообщения без кода самой команды

```

```

        creatmsk tra
        @echo $(BEEP) #подает звуковой сигнал,
#                               привлекая внимание
#                               пользователя

enhncemsk:    enhncemsk.c
              $(CC) $(CFLAGS) enhncemsk.c -o enhncemsk
              @echo $(BEEP)

mask_pre:    mask_pre.c
              $(CC) $(CFLAGS) mask_pre.c -o mask_pre
              @echo $(BEEP)

Int:         *.c    #проверка файлов на ошибочное
#                               использование типов
              $(LINT) *.c
              touch Int
              @echo $(BEEP)

print:       $(FILES)    #напечатать тексты файлов
#                               с последними исправлениями
#                               (отмеченные знаками $?)
              pr $? | lpr #выходные данные при
#                               выполнении команды pr
#                               пересылаются через канал "|" #
на принтер lpr
#                               для распечатки
              touch print
              @echo $(BEEP)

```

Файлы `Int` и `print` являются нулевыми файлами, используемыми для определения момента времени, когда исходные тексты последний раз проверялись на наличие ошибок и печатались.

Команда

`make имя`

обеспечивает выполнение команд, ассоциированных с именем *имя*, если файл *имя* содержит старую версию текста. Если имя файла в команде `make` не указано, то выбирается первое имя из файла `makefile`. Ниже перечислены команды, с помощью которых поддерживается соответствие листингов программ и состояния разрабатываемой системы, а также соответствие всех компонентов системы:

| Команда | Действие команды |
|----------------|---|
| make | Обеспечивает соответствие состояния программы formI тому состоянию, которое она должна иметь к данному моменту времени. |
| make formI | То же, что и для вышеприведенной команды, так как formI является первым именем. |
| make tra.mask | Создает маску для формирования файлов типа tra. |
| make enhncemsk | Обеспечивает соответствие состояния программы enhncemsk тому состоянию, которое она должна иметь к данному моменту времени. |
| make Int | Обеспечивает соответствие состояния файла Int тому состоянию, которое он должен иметь к данному моменту времени, и обеспечивает использование программы lint для всех программ на языке Си. |
| make print | Печатает тексты всех модифицированных файлов. |

Рассмотрим теперь работу программы make подробнее. Она вызывается командой

```
make [ -f makefile ] [ опция ] [ имена ]
```

Программа make выполняет все команды, записанные в файле *makefile*, для внесения изменений в один или более файлов, указанных в поле *имена*. Эти имена обычно являются именами программ, которые указаны в файле *makefile* вместе с именами других файлов, от которых эти программы зависят. При отсутствии опции -f ищутся файлы makefile или Makefile в указанном порядке. Если в качестве имени файла *makefile* указан знак "-", то входные данные считываются из файла стандартного ввода. Программа make позволяет внести коррекцию в такой файл, который зависит от ранее модифицированных файлов или отсутствует в библиотеке.

Файл makefile содержит последовательность элементов, описывающих зависимости между файлами. Первая строка элемента является списком тех файлов, которые предполагается корректировать явным указанием (имена этих файлов разделены пробелами), затем следует знак ":", после которого перечисляются файлы, от которых зависят файлы первого списка. Текст после знака ";" и все последующие строки, начинающиеся с отступа, являются командами операционной системы, обеспечиваю-

щими коррекцию файлов. Если некоторое имя файла появляется слева в более чем одной строке со знаком ":", то такой файл зависит от всех файлов, стоящих правее знака ":" во всех таких строках, но для него может быть указана только одна последовательность команд. При наличии знака "::" вместо двосточия последовательность команд после этой строки выполняется только в том случае, когда дата коррекции файла не соответствует текущему состоянию файлов, имена которых указаны справа от знака "::"; эти команды не относятся к другим строкам со знаками ":", в которых может быть записано имя данного файла.

Комментарии начинаются знаком # и заканчиваются концом строки. Ниже приведены некоторые возможные значения опции:

- p Трассировка и печать, но команды коррекции файлов не выполняются.
- t Обновление даты последней коррекции файла без выполнения каких-либо команд.

Приложение В

Стандарт ANSI языка Си

Процесс стандартизации языка Си продолжается. Технический комитет ANSI (рабочая группа X3J11) работает сейчас над стандартом языка Си. Большинство изменений, внесенных в язык, уже отражено в работе [77]. Некоторые из этих изменений приведены ниже.

| | |
|-------------------------------|---|
| Ключевые слова | Добавлено ключевое слово <code>const</code> и исключено ключевое слово <code>entry</code> . |
| Знаковые константы | Разрешено использование многозначных констант. |
| Строки | Соседние строковые литералы объединяются в один строковый литерал. |
| Арифметические преобразования | Вычисления с плавающей точкой могут выполняться с одинарной точностью. |
| Типы данных | Добавлен тип <code>const</code> . Определена эквивалентность типов; два типа эквивалентны, если эквивалентны типы их компонентов. |

| | |
|-----------------------------|---|
| Операторы | Результатом выполнения оператора <code>sizeof</code> является беззнаковая целая константа соответствующей разрядности. |
| Выражения | Добавлен унарный плюс. Указатели различных типов не могут употребляться одновременно. |
| Описания | Модификатор типа можно использовать с ключевыми словами <code>long</code> , <code>short</code> и <code>char</code> . В описаниях функций могут быть указаны типы формальных параметров функций. |
| Инициализация | Разрешена инициализация агрегированных данных типа <code>automatic</code> . |
| Внешние определения функций | Формальные параметры типа <code>float</code> не могут автоматически преобразовываться к типу <code>double</code> . Имя функции может использоваться в качестве формального параметра. |
| Главная программа | Каждая программа, выполняемая под управлением операционной системы, должна иметь функцию с именем <code>main</code> . В каждой программе эта функция будет выполняться первой. |
| Препроцессор | В инструкциях препроцессора до или после знака <code>#</code> может располагаться любое число пробелов. Добавлен унарный оператор <code>defined</code> . Добавлена директива <code>#elif</code> . |

Приложение Г

Таблицы кодов ASCII

Восьмеричное представление

| | | | | | | | |
|---------|---------|---------|---------|---------|---------|---------|---------|
| 000 nul | 001 soh | 002 stx | 003 etx | 004 eot | 005 enq | 006 ack | 007 bel |
| 010 bs | 011 ht | 012 nl | 013 vt | 014 np | 015 cr | 016 so | 017 si |
| 020 dle | 021 dc1 | 022 dc2 | 023 dc3 | 024 dc4 | 025 nak | 026 syn | 027 etb |
| 030 can | 031 em | 032 sub | 033 esc | 034 fs | 035 gs | 036 rs | 037 us |
| 040 sp | 041 ! | 042 " | 043 # | 044 \$ | 045 % | 046 & | 047 ' |
| 050 (| 051) | 052 * | 053 + | 054 , | 055 - | 056 . | 057 / |
| 060 0 | 061 1 | 062 2 | 063 3 | 064 4 | 065 5 | 066 6 | 067 7 |
| 070 8 | 071 9 | 072 : | 073 ; | 074 < | 075 = | 076 > | 077 ? |
| 100 @ | 101 A | 102 B | 103 C | 104 D | 105 E | 106 F | 107 G |
| 110 H | 111 I | 112 J | 113 K | 114 L | 115 M | 116 N | 117 O |
| 120 P | 121 Q | 122 R | 123 S | 124 T | 125 U | 126 V | 127 W |
| 130 X | 131 Y | 132 Z | 133 [| 134 \ | 135] | 136 ^ | 137 _ |
| 140 ` | 141 a | 142 b | 143 c | 144 d | 145 e | 146 f | 147 g |
| 150 h | 151 i | 152 j | 153 k | 154 l | 155 m | 156 n | 157 o |
| 160 p | 161 q | 162 r | 163 s | 164 t | 165 u | 166 v | 167 w |
| 170 x | 171 y | 172 z | 173 { | 174 | 175 } | 176 ~ | 177 del |

Шестнадцатеричное представление

| | | | | | | | |
|--------|--------|--------|--------|--------|--------|--------|--------|
| 00 nul | 01 soh | 02 stx | 03 etx | 04 eot | 05 enq | 06 ack | 07 bel |
| 08 bs | 09 ht | 0a nl | 0b vt | 0c np | 0d cr | 0e so | 0f si |
| 10 dle | 11 dc1 | 12 dc2 | 13 dc3 | 14 dc4 | 15 nak | 16 syn | 17 etb |
| 18 can | 19 em | 1a sub | 1b esc | 1c fs | 1d gs | 1e rs | 1f us |
| 20 sp | 21 ! | 22 " | 23 # | 24 \$ | 25 % | 26 & | 27 ' |
| 28 (| 29) | 2a * | 2b + | 2c , | 2d - | 2e . | 2f / |
| 30 0 | 31 1 | 32 2 | 33 3 | 34 4 | 35 5 | 36 6 | 37 7 |
| 38 8 | 39 9 | 3a : | 3b ; | 3c < | 3d = | 3e > | 3f ? |
| 40 @ | 41 A | 42 B | 43 C | 44 D | 45 E | 46 F | 47 G |
| 48 H | 49 I | 4a J | 4b K | 4c L | 4d M | 4e N | 4f O |
| 50 P | 51 Q | 52 R | 53 S | 54 T | 55 U | 56 V | 57 W |
| 58 X | 59 Y | 5a Z | 5b [| 5c \ | 5d] | 5e ^ | 5f _ |
| 60 ' | 61 a | 62 b | 63 c | 64 d | 65 e | 66 f | 67 g |
| 68 h | 69 i | 6a j | 6b k | 6c l | 6d m | 6e n | 6f o |
| 70 p | 71 q | 72 r | 73 s | 74 t | 75 u | 76 v | 77 w |
| 78 x | 79 y | 7a z | 7b { | 7c | 7d } | 7e ~ | 7f del |

Приложение Д

Характеристики компилятора, зависящие от реализации

Набор знаков

| Типы ЭВМ | DEC PDP-11 | DEC VAX-11 | AT & T 3B | IBM 370 | Motorola 68000 |
|----------|------------|------------|-----------|---------|----------------|
| Стандарт | ASCII | ASCII | ASCII | EBCDIC | ASCII |

Память (в битах), обычно отводимая компиляторами языка Си на различных ЭВМ для основных типов переменных

| Типы ЭВМ | DEC PDP-11 | DEC VAX-11 | AT & T 3B | IBM 370 | Motorola 68000 |
|-----------|------------|------------|-----------|---------|----------------|
| char | 8 | 8 | 8 | 8 | 8 |
| int | 16 | 32 | 32 | 32 | 8 или 16 |
| short | 16 | 16 | 16 | 16 | 16 |
| long | 32 | 32 | 32 | 32 | 32 |
| float | 32 | 32 | 32 | 32 | 32 |
| double | 64 | 64 | 64 | 64 | 32 или 64 |
| Указатели | 16 | 32 | 32 | 32 | 32 |

Список литературы

1. Ada 1983. Reference Manual for the Ada Programming Language. United States Department of Defense (January).
2. Anderson, B. 1980. Type Syntax in the Language C: an object lesson in syntactic innovation. SIGPLAN Notices, v15, no. 3 (March).
3. AT&T UNIX (Release 5.0) 1982. UNIX System User's Manual (Release 5.0). AT&T Bell Laboratories, Murray Hill, N.J. 07974.
4. AT&T UNIX (System V) 1983. UNIX System V Reference Manuals. AT&T Technologies, 1983.
5. Berkeley UNIX 1981. UNIX Programmer's Manual (4.1 BSD). Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.
6. Bourne, S. R. 1982. The UNIX System. Addison-Wesley Publishing Co.
7. Bowles, K. L. Problem Solving Using PASCAL. Springer-Verlag.
8. Brinch Hansen, P. 1973. Concurrent Programming Concepts. ACM Computing Surveys, v6, no. 4 (December), pp.223-245.
9. Brinch Hansen, P. 1975. The Programming Language Concurrent Pascal. IEEE Transactions on Software Engineering, vSE-1, no. 2 (June).
10. Byte Magazine 1983. Special Issue of Byte magazine on the C programming language, v8, no. 8 (August).

11. Christian, K. 1983. The UNIX Operating System. John Wiley.

[Имеется перевод: Кристиан К. Операционная система UNIX: Пер. с англ. - М.: Финансы и статистика, 1985. - 318 с.]

12. Collinson, R. P. A. 1981. Comments on Style in C. UKC Computing Lab-7, Computing Lab, Kent University, Canterbury, England.

13. Dahl, O. J., E. W. Dijkstra and C. A. R. Hoare 1972. Structured Programming. Academic Press.

[Имеется перевод: Дал У., Дейкстра Э., Хоар К. Структурное программирование: Пер. с англ. - М.: Мир, 1975.- 248 с.]

14. Dijkstra, E. W. 1968. Goto Statement Considered Harmful. CACM, v11 (March), pp. 147-148.

15. Dijkstra, E. W. 1972. Notes on Structured Programming. In Structured Programming edited by O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, Academic Press.

16. Dijkstra, E. W. 1976. A Discipline of Programming. Prentice-Hall.

17. Evans Jr., A. 1984. A Comparison of Programming Languages: Ada, Pascal, C. In Comparing and Assessing Programming Languages edited by Alan Feuer and Narain Gehani, Prentice-Hall [Feuer and Gehani 1984].

18. Feldman, S. I. 1979. Make - A Program for Maintaining Computer Programs. Software - Practice and Experience, v9, pp. 255-265.

19. Feuer, A. 1982. C Puzzle Book. Prentice-Hall.

[Имеется перевод: Керниган Б., Ритчи Д., Фьюэр А. Язык программирования Си. Задачи по языку Си: Пер. с англ. - М.: Финансы и статистика, 1985. - 279 с.]

20. Feuer, A. and N. Gehani 1982. A Comparison of Programming Languages C and Pascal. ACM Computing Surveys, v14, no. 1 (March), pp. 73-92.

21. Feuer, A. and N. Gehani (Editors) 1984. Comparing and Assessing Programming Languages. Prentice-Hall.

[Имеется перевод: Языки программирования Ада, Си, Паскаль. Сравнение и оценка: Пер. с англ./Под ред. А. Фьюэра, Н. Джехани. - М.: Радио и связь, 1989]

22. Fitzhorn, P. A. and G. R. Johnson 1981. C: Toward a Concise Syntactic Description. SIGPLAN Notices, v16, no. 12 (December), pp. 14-21.

23. Gannon, J. D. 1975. Language Design to Enhance Program Reliability. Technical Report CSRG-47, University of Toronto.

24. Gannon, J. D. 1977. An Experimental Evaluation of Data Type Conventions. CACM, v20, no. 8 (August).

25. Gehani, N. 1977. Units of Measure as Data Attribute, Computer Languages, v2, pp. 93-111.

26. Gehani, N. H. 1981. Program Development by Stepwise Refinement and Related Topics. Bell System Technical Journal, v60, no.3 (March), pp. 347-378.

27. Gehani, N. H. 1983a. An Electronic Form System: An Experience in Prototyping. Software - Practice and Experience, v13, pp. 479-486.

28. Gehani, Narain 1983b. Ada: An Advanced Introduction Including Reference Manual for the Ada Language. Prentice-Hall.

29. Gries, D. 1976. An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs. IEEE Transactions on Software Engineering v2, no. 4, pp 238-243.

30. Gries, D. 1979. cand and cor before and then or else in Ada. Technical Report TR79-402, Department of Computer Science, Cornell University, Ithaca, N. Y. 14853.

31. Halfant, M. 1983. The UNIX C Compiler in CP/M Environment. Byte, v8, no. 8 (August), pp. 243-267.
32. Hamming, R. W. 1973. Numerical Methods for Scientists and Engineers (second edition). McGraw Hill.
33. Hancock, L. and M. Krieger 1982. The C Primer. McGraw Hill.
- [Имеется перевод: Хэнкок Л., Кригер М, Введение в программирование на языке Си: Пер. с англ. - М.: Радио и связь, 1986. - 192 с.]
34. Harbison, S. P. and G. L. Steele, Jr. 1983. The C Language Reference Manual. Tartan Labs.
35. Hoare, C. A. R. 1962. Quicksort. Computer Journal, v5, no. 1, pp. 10-15.
36. Hoare, C. A. R. 1973. Hints on Programming Language Design ACM SIGACT/SIGPLAN Symposium of Principles on Programming Languages (October), Boston, Mass.
37. Hoare, C. A. R. and N. Wirth 1973. An Axiomatic Definition of the Programming Language Pascal, Acta Informatica v2, pp. 335-355.
38. Hoare, C. A. R. 1978. Towards a Theory of Parallel Programming Methodology, a Collection of Articles by Members of WG2.3 edited by D. Gries, Springer-Verlag.
39. Horowitz, E. 1983. Fundamentals of Programming Languages. Computer Science Press.
40. Horning, J. J. 1979 Effects of Programming Languages on Reliability. In Computer Systems Reliability edited by T. Anderson and B. Randell, Cambridge University Press.
41. Houston, J., J. Broderick and L. Kent 1983. Comparing C Compilers for CP/M-86. Byte, v8, no. 8 (August), pp. 82-106.
42. Pascal 1980. Second Draft Proposal of the ISO Pascal Standard (January 1981). Pascal News, no. 20.

43. Jensen. K. and N. Wirth 1974. The Pascal User Manual and Report. Springer-Verlag.

[Имеется перевод: Йенсен К., Вирт Н. Паскаль: Руководство для пользователя: Пер. с англ. - М.: Финансы и статистика. - 1989]

44. Johnson, S. C. and B. W. Kernighan 1973. The Programming Language B. Unpublish notes.

45. Johnson, S. C. 1978a. A Portable Compiler: Theory and Practice. Fifth Annual ACM Symposium on Principles of Programming Languages (January), Tucson, Arizona.

46. Johnson, S. C. 1978b. Lint, a C Program Checker.

47. Johnson, S. C. and B. W. Kernighan 1973. The C Language and Models for Systems Programming. Byte, v8, no. 8 (August), pp. 48-60.

48. Joyce, J. 1983a. A C Language Primer, Part 1: Constructs and Conventions. Byte, v8, no. 8 (August), pp. 64-78.

49. Joyce, J. 1983b. A C Language Primer, Part 2: Tool Building in C. Byte, v8, no. 9 (September), pp.289-302.

50. Katzenelson, J. 1983a. Introduction to Enhanced C. Software - Practice and Experience, v13, no. 7 (July), pp. 551-576.

51. Katzenelson, J. 1983a. Higher Level Programming and Data Abstractions - A Case Study Using Enhanced C. Software - Practice and Experience, v13, no. 7 (July), pp. 577-595.

52. Kernighan, B. W. and P. J. Plauger 1974. The Elements of Programming Style. McGraw-Hill.

[Имеется перевод: Керниган Б., Пладжер Ф. Элементы стиля программирования: Пер. с англ. - М.: Радио и связь, 1984. - 160 с.]

53. Kernighan, B. W. and P. J. Plauger 1976. Software Tools. Addison-Wesley.

54. Kernighan, B. W. and P. J. Plauger 1981. Software Tools in Pascal. Addison-Wesley.

[Имеется перевод: Керниган Б., Плоджер Ф. Инструментальные средства программирования на языке Паскаль: Пер. с англ. - М.: Радио и связь, 1985. - 312 с.]

55. Kernighan, B. W. and Ritchie 1978. The C Programming Language. Prentice-Hall.

[Имеется перевод: Керниган Б., Ритчи Д., Фьюэр А. Язык программирования Си. Задачи по языку Си: Пер. с англ. - М.: Финансы и статистика, 1985. - 279 с.]

56. Kernighan, B. W. and M. D. McIlroy (Editors) 1979. UNIX Programmer's Manual, Seventh Edition. AT&T Bell Laboratories, Murray Hill, N. J. 07974.

57. Kern, C. O. 1983. Five C Compilers for CP/M-80. Byte, v8, no. 8 (August), pp. 110-130.

58. Kernighan, B. W. and R. Pike 1984. The UNIX Programming Environment. Prentice-Hall.

59. Lalonde, W. R. and J. R. Pugh 1983. A Simple Technique for Converting from Pascal Shop to a C Shop. Software - Practice and Experience, v13, pp. 771-775.

60. Lear, E. 1964. The Nonsense Books of Edward Lear. The New American Library.

61. Lee, P. A. 1983. Exception Handling in C Programs. Software - Practice and Experience, v13, no. 5 (May), pp. 389-405.

62. Linhart, J. 1983. Managing Software Development with C. Byte, v8, no.8 (August), pp. 172-182.

63. Liskov, B. and Zilles 1974. Programming with Abstract Data Types. SIGPLAN Notices, v9, no. 4 (April).

64. Liskov, B. 1976. Discussion in the Design and Implementation of Programming Languages edited by John H. Williams and D. A. Fisher, p. 25, Springer-Verlag.

65. Mateti, P. 1979. Pascal versus C: A Subjective Comparison, Proceedings of the Symposium on Language Design and Programming Methodology (September).

66. McGettrick, A. D. and P. D. Smith 1983. Graded Problems in Computer Science. Addison-Wesley.

67. McIlroy, M. D. 1960. Macro Instruction Extension of Compiler Languages. CACM, v3, no. 4, pp. 414-420.

68. Morris, J. H., Jr. 1973. Types are not Sets. ACM Symposium on Principles of Programming Languages, Boston, MA.

69. Naur, P. 1963. Revised Report on the Algorithmic Language ALGOL 60. CACM, v6, no. 1 (January), pp. 1-17.

70. Phraner, R. A. 1983. Nine C Compilers for the IBM PC. Byte, v8, no. 8 (August), pp. 134-168.

71. Plum, T. 1983. Learning to Program in C. Plum Hall, Cardiff, N. J.

72. Pratt, T. W. 1975. Programming Languages: Design and Implementation. Prentice-Hall.

73. Pratt, V. 1983. Five Paradigm Shifts in Programming Language Design and Their Realization in Viron, a Dataflow Programming Environment. Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages (January), Austin, TX.

74. Purdum, J. C. 1983. C Programming Guide. Que Corporation, Indianapolis, IN.

75. Richards, M. 1969. BCPL: Tool for Compiler Writing and Systems Programming. Proc. AFIPS SICC, v34, pp. 557-566.

76. Ritchie, D. M., S. Johnson, M. E. Lesk and B. W. Kernighan 1978. The C Programming Language. Bell System Technical Journal, Part 2, v57, no. 6 (July-August), pp. 1991-2019.

77. Ritchie, D. M. 1980. The C Programming Language - Reference Manual. AT&T Bell Laboratories, Murray Hill, N. J. 07974.

78. Schwartz, J. T. 1975. On Programming - Interim Report of the SETL Project. Courant Institute of Mathematical Sciences, NY.

79. Sethi, R. 1980. A Case Study in Specifying the Semantics of a Programming Language. Seventh Annual ACM Symposium on Principles of Programming Languages (January), Las Vegas, NV.

80. Sethi, R. 1981. Uniform Syntax for Type Expressions and Declarations. Software - Practice and Experience, v13, no. 6 (June), pp. 623-628.

81. Stroustrup, B. 1982. A Set of C Classes for Coroutine-Style Programming. Computing Science Technical Report No. 90, AT&T Bell Laboratories, Murray Hill, N. J. 07974.

82. Stroustrup, B. 1983. Adding Classes of the C Language: An Exercise in Language Evolution. Software - Practice and Experience, v13, pp. 139-161.

83. Stroustrup, B. 1984a. Data Abstraction in C. To be published.

84. Stroustrup, B. 1984b. The C++ Programming Language - Reference Manual. Computing Science Technical Report No. 108, AT&T Bell Laboratories, Murray Hill, N. J. 07974.

85. Van Wyk, C. J., J. L. Bentley and P. J. Weinberger 1982. Efficiency Considerations for C Programs on VAX 11/780. Technical Report CMU-CS-82-134, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

86. Welsh, J., M. J. Sneeringer and C. A. R. Hoare 1977. Ambiguities and Insecurities in Pascal. Software - Practice and Experience, v7, no. 6 (November), pp. 685-696.

87. Wirth, N. and C. A. R. Hoare 1966. A Contribution to the Development of ALGOL. CACM, v9, no. 6 (June), pp. 413-433.

88. Wirth, N. 1971a. The Design of a Pascal Compiler. Software - Practice and Experience, v1, pp. 309-333.

89. Wirth, N. 1971b. The Programming Language Pascal. Acta Informatica, v1, pp. 35-63.

90. Wirth, N. 1971c. Program Development by Stepwise Refinement. CACM, v14, no. 4, pp. 221-226.

91. Wirth, N. 1973. Systematic Programming: An Introduction. Prentice-Hall.

[Имеется перевод: Вирт М. Систематическое программирование. Введение: Пер. с англ. - М.: Мир, 1977. - 184 с.]

92. Wirth, N. 1975. An Assessment of the Programming Language Pascal. Proceedings - 1975 International Conference on Reliable Software (April), Los Angeles, CA.

93. Wulf, W. and M. Shaw 1973. Global Variable Considered Harmful. SIGPLAN Notices, v8, n2, pp. 28-34.

94. Zahn, C. T. 1979. C Notes: A Guide to the C Programming Language. Yourdon Press, NY.

Оглавление

| | |
|--|-----------|
| Предисловие | 5 |
| Глава 1. Введение и основные понятия..... | 8 |
| 1.1 Пример программы на языке Си..... | 8 |
| 1.1.1 Компиляция и выполнение программы калькулятора в системе UNIX | 14 |
| 1.2. Основные понятия..... | 16 |
| 1.2.1. Алфавит | 16 |
| 1.2.2. Идентификаторы | 16 |
| 1.2.3. Литералы | 17 |
| 1.2.4. Комментарии | 20 |
| 1.2.5. Точка с запятой - признак конца оператора | 20 |
| 1.3. Константы | 20 |
| 1.4. Задачи | 21 |
| Глава 2. Типы и переменные..... | 22 |
| 2.1. Основные типы..... | 23 |
| 2.1.1. Знаки | 23 |
| 2.1.2. Целые | 23 |
| 2.1.3. Перечисляемые типы..... | 24 |
| 2.1.4. Булевы, или логические значения | 26 |
| 2.1.5. Плавающая точка..... | 27 |
| 2.1.6. Тип void (пустой)..... | 27 |
| 2.2. Производные типы..... | 28 |
| 2.2.1. Массивы..... | 28 |
| 2.2.2. Структуры | 29 |
| 2.2.3. Объединения | 34 |
| 2.2.4. Переменные структуры..... | 35 |
| 2.2.5. Указатели | 38 |
| 2.3. Описания типа..... | 45 |
| 2.4. Определения и описания | 46 |
| 2.4.1. Классы памяти..... | 46 |
| 2.4.2. Типы данных..... | 49 |
| 2.4.3. Описатели | 49 |
| 2.4.4. Примеры определений и описаний объектов | 50 |
| 2.4.5. Инициализаторы | 52 |
| 2.4.6. Комментарии к синтаксису | |

| | |
|---|-----------|
| описаний и определений | 53 |
| 2.4.7. Примеры, иллюстрирующие использование эквивалентностей..... | 55 |
| 2.4.8. Использование оператора <i>typedef</i> для упрощения понимания описаний и определений..... | 56 |
| 2.4.9. Заключительные замечания об описаниях и определениях..... | 57 |
| 2.4.10. Эквивалентность типов..... | 57 |
| 2.5. Преобразования типов..... | 58 |
| 2.5.1. Неявное преобразование типа | 58 |
| 2.5.2. Арифметические преобразования..... | 59 |
| 2.5.3. Явные преобразования типов | 60 |
| 2.6. Задачи | 61 |
| Глава 3. Операторы и выражения..... | 63 |
| 3.1. Операторы | 63 |
| 3.1.1. Операторы вызова функции, индексирования и выбора..... | 63 |
| 3.1.2. Унарные операторы..... | 64 |
| 3.1.3. Мультипликативные операторы..... | 66 |
| 3.1.4. Аддитивные операторы..... | 67 |
| 3.1.5. Операторы сдвига..... | 68 |
| 3.1.6. Операторы отношения..... | 69 |
| 3.1.7. Операторы равенства и неравенства | 69 |
| 3.1.8. Оператор поразрядное <i>и</i> | 70 |
| 3.1.9. Оператор поразрядное <i>исключающее или</i> | 70 |
| 3.1.10. Оператор поразрядное <i>включающее или</i> | 71 |
| 3.1.11. Логический (условный) оператор <i>и</i> | 71 |
| 3.1.12. Логический (условный) оператор <i>или</i> | 71 |
| 3.1.13. Условный оператор..... | 72 |
| 3.1.14. Операторы присваивания..... | 73 |
| 3.1.15. Оператор запятой..... | 75 |
| 3.1.16. Таблица приоритетов и порядка выполнения операторов | 76 |
| 3.2. Выражения | 77 |
| 3.2.1. Постоянные выражения..... | 77 |

| | |
|---|-----------|
| 3.3. Задачи | 78 |
| Глава 4. Операторы управления | 79 |
| 4.1. Выражения и операторы..... | 79 |
| 4.2. Пустой оператор..... | 79 |
| 4.3. Составной оператор..... | 80 |
| 4.4. Оператор присваивания..... | 80 |
| 4.5. Оператор <i>if</i> | 81 |
| 4.6. Оператор <i>switch</i> | 83 |
| 4.7. Циклы | 85 |
| 4.7.1. Цикл <i>while</i> | 85 |
| 4.7.2. Цикл <i>for</i> | 85 |
| 4.7.3. Цикл <i>do</i> | 86 |
| 4.4.8. Оператор <i>break</i> | 87 |
| 4.9. Оператор <i>continue</i> | 87 |
| 4.10. Оператор вызова функции | 88 |
| 4.11. Оператор <i>return</i> | 88 |
| 4.12. Оператор <i>goto</i> | 88 |
| 4.13. Метки операторов..... | 89 |
| 4.14. Задачи | 89 |
| Глава 5. Функции и завершенные программы..... | 90 |
| 5.1. Функции | 90 |
| 5.1.1. Описания параметров..... | 92 |
| 5.1.2. Управление видимостью функций | 92 |
| 5.1.3. Вызов функций..... | 93 |
| 5.1.4. Обращение к функции до ее определений..... | 94 |
| 5.1.5. Пример, иллюстрирующий передачу параметров..... | 96 |
| 5.1.6. Автоматические преобразования фактических параметров..... | 97 |
| 5.1.7. Передача функций в качестве параметров..... | 98 |
| 5.1.8. Спецификации функции..... | 99 |
| 5.2. Лексическая область действия идентификаторов | 100 |
| 5.3. Ввод и вывод..... | 101 |
| 5.3.1. Использование потоков, определяемых программистом..... | 104 |
| 5.4. Переназначение ввода и вывода (в системе UNIX) | 105 |

| | |
|---|-----|
| 5.5. Головные программы | 107 |
| 5.6. Примеры | 109 |
| 5.6.1. Удаление последовательностей знаков форматирования из текста программы на языке Ада..... | 109 |
| 5.6.2. Получение изображения бланка из табличного описания | 113 |
| 5.6.3. Улучшение изображения документа на терминале HP2621 | 114 |
| 5.6.4. Функция <i>sine</i> вычисления значения синуса..... | 116 |
| 5.6.5. Метод наименьших квадратов для построения кривой [32]..... | 117 |
| 5.6.6. Быстрая сортировка..... | 120 |
| 5.6.7. Управление скоростью автомобиля | 123 |
| 5.6.8. Передача функций в качестве параметров..... | 126 |
| 5.6.9. Поиск в массиве..... | 128 |
| 5.7. Задачи | 130 |

Глава 6. Раздельная компиляция и

| | |
|---|------------|
| абстрагирование данных..... | 132 |
| 6.1. Область действия внешних определений и описаний | 133 |
| 6.2. Раздельная компиляция..... | 134 |
| 6.3. Абстрактные типы данных и сокрытие информации..... | 135 |
| 6.3.1. Стек - пример абстрактного типа данных..... | 135 |
| 6.3.2. Ограниченность файлов как средств абстрагирования данных | 138 |
| 6.4. Классы | 138 |
| 6.4.1. Конструкторы, деструкторы и расширение операторов и функций в языке Си++ | 142 |
| 6.4.2. Заключительные замечания о классах..... | 143 |
| 6.5. Примеры | 143 |
| 6.5.1. Таблица символов..... | 143 |
| 6.5.2. Функции для обработки списков | 146 |
| 6.5.3. Класс <i>buffer</i> | 149 |
| 6.6. Задачи | 151 |

| | |
|---|------------|
| Глава 7. Особые ситуации | 152 |
| 7.1. Различные сигналы..... | 154 |
| 7.2. Установка режима обработки особых ситуаций..... | 155 |
| 7.2.1. Пример использования сигналов для обработки ошибок при выполнении операций с плавающей точкой..... | 157 |
| 7.3. Генерация и посылка сигналов..... | 160 |
| 7.4. Примеры | 160 |
| 7.4.1. Программа калькулятора с обработкой особых ситуаций (вариант 1)..... | 160 |
| 7.4.2. Программа калькулятора с обработкой особых ситуаций (вариант 2)..... | 162 |
| 7.4.3. Программа калькулятора с обработкой особых ситуаций (вариант 3)..... | 165 |
| 7.5. Задачи | 167 |
| Глава 8. Параллельное программирование | 168 |
| 8.1. Параллельное программирование на языке Си в операционной системе UNIX | 169 |
| 8.2. Создание процесса с помощью библиотечной функции fork..... | 170 |
| 8.3. Библиотечная функция <code>exec1</code> для перекрывающихся процессов..... | 171 |
| 8.4. Каналы - синхронные средства связи | 172 |
| 8.4.1. Установка канала..... | 173 |
| 8.4.2. Переназначение стандартного ввода-вывода..... | 174 |
| 8.5. Примеры | 176 |
| 8.5.1. Асинхронное, или параллельное, копирование файла..... | 177 |
| 8.5.2. Подсчет неформатирующих знаков в файле | 179 |
| 8.6. Задачи | 183 |
| Глава 9. Препроцессор языка Си..... | 183 |
| 9.1. Макроопределения и макровыводы..... | 184 |
| 9.1.1. Макровыводы | 184 |
| 9.1.2. Простые макроопределения..... | 184 |
| 9.1.3. Параметризованные макроопределения | 185 |

| | |
|--|------------|
| 9.1.4. Определение констант | 186 |
| 9.1.5. Оперативная генерация кода | 187 |
| 9.1.6. Удаление макроопределений | 188 |
| 9.2. Включение файлов в программу | 188 |
| 9.3. Условная компиляция | 189 |
| 9.4. Заключительные замечания | 192 |
| 9.5. Пример использования препроцессора | 192 |
| 9.6. Задачи | 193 |
| Глава 10. Последний пример | 195 |
| 10.1. Получение информации из базы данных | 195 |
| 10.1.1. Стратегия разработки базы данных | 196 |
| 10.1.2. Функции базы данных | 197 |
| 10.2. Задачи | 202 |
| Приложение А. Некоторые библиотечные функции | 202 |
| Приложение Б. Некоторые инструментальные средства | 242 |
| Приложение В. Стандарт ANSI языка Си | 252 |
| Приложение Г. Таблицы кодов ASCII | 254 |
| Приложение Д. Характеристики компилятора, зависящие от реализации | 256 |
| Список литературы | 257 |

Производственное издание

Джехани Нарайян

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ СИ

Заведующая редакцией О. В. Толкачева

Редактор М. Г. Коробочкина

Художественный редактор Т. В. Бусарова

Переплет художника О. С. Белова

Технический редактор А. Н. Золотарева

ИБ № 1619

Подписано в печать с оригинала-макета 29.11.88. Формат 60х88/16. Бумага офс. № 2.
Гарнитура "Таймс". Печать офсетная. Усл. печ. л. 16,66. Усл. кр.-отт. 16,66.
Уч.-изд. л. 12,74. Тираж 20 000 экз. Изд. № 22185. Заказ № 1728. Цена 1 р. 30 к.
Издательство "Радио и связь". 101000 Москва, Почтамт, а/я 693

Московская типография № 4 Союзполиграфпрома при Государственном комитете
СССР по делам издательств, полиграфии и книжной торговли.
129041 Москва, Б. Переяславская ул., д. 46

